
How to Become a Software QA Tester

Version 1.0

Roman Savenkov
www.qatutor.com

Copyright 2008 Roman Savenkov

All rights reserved

Published by Roman Savenkov

Address:

P.O. Box 2785
Sunnyvale, CA, 94087

Email:

roman@qatutor.com

Web:

www.qatutor.com



Roman Savenkov is a software testing practitioner and educator specializing in the topics of software testing and career development.

His mission statement:

**I help my clients to improve their software;
I help my students to improve their lives.**

Many thanks to...

Alex Babichev, Alex Khatilov, Alexander Gurevich, Alexander Lobach, Alexey Bulat, Anitha Karpagaganapathy, Ann Nevero, Cameron Bigger, Cem Kaner, Claudia Volkman (editor), Daniel Kionka, Eing Ong, Ilya Dunin, Irene Wang, James Bach, James Kellas, Jason Fisher, Jennifer Hu, Jerry Tso, John Canicosa, John Milton Fogg, John T. Reed, Katrina Glerum, Katyusha Fogg, Marina Latysh, Max Levchin, Mike Kelley, Nikita Toulinov, Oxana Wootton, Peter Shvets, Prashant Nedungadi, Rajeev Raman, Samira Basha, Scirocco Michelle Six, Sergey Korsun (caricatures), Sergey Martynenko, Stephen Ostrowski, Steve Mui, Tho Dao, Vadim Kotlyar, Vlod Kalicun, Wendy Yu, Yury Brodskiy, Yury Luiso, Zoya Savenkova.

Table of Contents (short story)

Introduction.....	1
Disclaimer.....	5
Lecture 1. A Bug or Not a Bug.....	9
Quick Intro.....	10
Three Conditions Of A Bug's Existence.....	11
The Gist Of Testing.....	11
Spec, Spec, Spec.....	12
Software Bugs And Spec Bugs.....	13
Other Sources Of Expected Results.....	14
Why We Call Them "Bugs"?.....	17
Lecture Recap.....	17
Questions & Exercises.....	18
Lecture 2. The Purpose of Testing.....	19
Quick Intro.....	20
Exposing Misconceptions.....	20
How To Make Use Of Stats For Post-Release Bugs.....	26
Testing And QA.....	28
Lecture Recap.....	29
Questions & Exercises.....	29
Lecture 3. Test Cases and Test Suites.....	31
Quick Intro.....	32
Test Case Structure.....	32
Results Of The Test Case Execution.....	34
Useful Attributes Of The Test Case.....	35
Data-Driven Test Cases.....	38
Maintainability Of Test Cases.....	38
The Number Of Expected Results Inside One Test Case.....	42
Bad Test Case Practices.....	45
Test Suites.....	51
States Of A Test Case.....	59
The Importance Of Creativity.....	60
Three Important Factors That Affect Test Cases.....	61

Checklists.....	62
Lecture Recap.....	63
Questions & Exercises.....	64
Lecture 4. The Software Development Life Cycle.....	66
Quick Intro.....	67
Idea.....	68
Product Design.....	71
Coding.....	87
Testing And Bug Fixes.....	111
Release.....	113
The Big Picture Of The Cycle.....	134
Lecture Recap.....	137
Questions & Exercises.....	142
Lecture 5. The Software Testing Life Cycle.....	144
Quick Intro.....	145
Research.....	146
Test Planning.....	148
Test Execution.....	149
Test Education And Reality.....	150
Lecture Recap.....	150
Questions & Exercises.....	151
Lecture 6. Classifying the Most Common Types of Testing.....	152
Quick Intro.....	153
By Knowledge Of The Internals Of The Software.....	154
By The Object Of Testing.....	162
By Time Of Test Execution.....	165
By Positivism Of Test Scenarios.....	166
By Degree Of Isolation Of Tested Components.....	168
By Degree Of Automation.....	176
By Preparedness.....	177
Lecture Recap.....	178
Questions & Exercises.....	181
Lecture 7. Test Preps.....	183
Quick Intro.....	184
The Tester's Mental Attitude.....	185
Intro To Special Skills In Bug Finding.....	186
Black Box Testing Techniques.....	187
When You Start To Implement Testing Techniques.....	210
Lecture Recap.....	210
Questions & Exercises.....	211
Lecture 8. Bug Tracking.....	212
Quick Intro.....	213
The Bug Tracking System.....	213

Bug Tracking Procedure.....	251
Quick Closing Note About BTS And BTP.....	255
Lecture Recap.....	255
Questions & Exercises.....	257
Lecture 9. Test Execution: New Feature Testing.....	258
Quick Intro.....	259
Test Estimates.....	261
Entry/Exit Criteria.....	263
Test Plan.....	264
Aggressive Testing From Jason Fisher.....	268
Lecture Recap.....	269
Questions & Exercises.....	270
Lecture 10. Test Execution: Regression Testing.....	272
Quick Intro.....	273
How To Choose Test Suites For Regression Testing.....	273
How To Resolve The Contradiction Between Our Limited Resources And The Ever-Growing Number Of Test Suites.....	278
Automation Of Regression Testing: Do It Right OR Forget About It.....	278
When Regression Testing Stops.....	296
Lecture Recap.....	297
Questions & Exercises.....	299
Lecture 11. How to Find Your First Job in Testing.....	301
Why You Have A REAL Chance To Find A Job In Testing.....	303
Mental Tuning.....	304
Job Hunting Activities.....	306
Lecture Recap.....	336
Questions & Exercises.....	338
Lecture 12. Bonus Tracks.....	339
How Much Technology Knowledge Must A Beginner Tester Have.....	340
What To Do If You Are The First Test Engineer At A Software Start-Up.....	341
What If You Are Asked To Do Test Automation As Your First Task.....	346
What Are Stock Options.....	346
How To Spot A Promising Start-up.....	349
Afterword.....	354
Links & Books.....	358
Glossary.....	360
Index.....	390

Table of Contents (long story)

Introduction.....	1
Disclaimer.....	5
Lecture 1. A Bug or Not a Bug.....	9
Quick Intro.....	10
Three Conditions Of A Bug's Existence.....	11
The Gist Of Testing.....	11
Spec, Spec, Spec.....	12
Software Bugs And Spec Bugs.....	13
Error Message During Registration.....	13
Other Sources Of Expected Results.....	14
Life Experience.....	14
Common Sense.....	15
Communication.....	16
Standards.....	16
Statistics.....	16
Valuable Opinion.....	17
Other Sources.....	17
Why We Call Them "Bugs"?.....	17
Lecture Recap.....	17
Questions & Exercises.....	18
Lecture 2. The Purpose of Testing.....	19
Quick Intro.....	20
Exposing Misconceptions.....	20
Misconception #1: "Testers Must Make Sure That 100% Of The Software Works Fine.".....	20
Froggy.py.....	20
Valid Input/Invalid Input.....	21
Input->Processing->Output.....	23
Misconception 2: "A Tester's Effectiveness Is Reflected In The Number Of Bugs Found Before The Software Is Released To Users.".....	25
How To Make Use Of Stats For Post-Release Bugs.....	26
Alan, Boris, Connor And 7 Bugs In Checkout.....	27
Testing And QA.....	28
Life Of Mr. G. From QA Perspective.....	28
Lecture Recap.....	29

Questions & Exercises.....	29
Lecture 3. Test Cases and Test Suites.....	31
Quick Intro.....	32
Checking Mr. Wilson's Backpack For Chips And Beer.....	32
Test Case Structure.....	32
Samuel Pickwick And Test Case With Credit Card: Scene 1.....	32
Samuel Pickwick And Test Case With Credit Card: Scene 2.....	33
Results Of The Test Case Execution.....	34
Why FAIL Is Good.....	34
Useful Attributes Of The Test Case.....	35
Unique ID/Priority/IDEA/SETUP And ADDITIONAL INFO/Revision History.....	35
Data-Driven Test Cases.....	38
Maintainability Of Test Cases.....	38
QA KnowledgeBase.....	42
The Number Of Expected Results Inside One Test Case.....	42
Log Files.....	44
Front End/Back End.....	45
Bad Test Case Practices.....	45
1. Dependency Between Test Cases.....	45
2. Poor Description Of The Steps.....	48
Trip To LA.....	48
3. Poor Description Of The Idea And/or Expected Result.....	50
Test Suites.....	51
First Test Case Execution.....	52
Author/Spec ID/PM/Developer/Priority/OVERVIEW/ GLOBAL SETUP AND ADDITIONAL INFO.....	53
Checkout With Credit Cards (TS7122).....	54
Checkout With AmEx (TS7131).....	56
Merging Test Suites.....	57
States Of A Test Case.....	59
Created/Modified/Retired.....	59
The Importance Of Creativity.....	60
Three Important Factors That Affect Test Cases.....	61
1. Importance Of The Project.....	61
2. Complexity Of The Project.....	62
3. Stage Of The Project.....	62
Checklists.....	62
Lecture Recap.....	63
Questions & Exercises.....	64
Lecture 4. The Software Development Life Cycle.....	66
Quick Intro.....	67
Idea.....	68
Once Upon A Time In California.....	68
Product Design.....	71
Breaking Rule #1: Clarity Of Details And Definitions.....	73
Breaking Rule #2: No Room For Misinterpretation.....	74
Breaking Rule #4: Solid, Logical Structure.....	75
Breaking Rule #5: Completeness.....	76
Breaking Rule #6: Compliance With Laws.....	76

Breaking Rule #7: Compliance With Business Practices.....	76
Spec Status.....	77
Spec Change Procedure.....	78
Examples/Flowcharts/Mock-ups.....	81
Coding.....	87
Typical Architecture Of Web Based Applications.....	87
1. Hire Good People, And Be Prepared To Give Them A Break.....	91
2. Direct, Fast, Effective Communication Between Coworkers.....	91
3. Code Inspections.....	93
4. Coding Standards.....	94
5. Realistic Schedules.....	95
6. Availability Of Documentation.....	95
7. Rules About Unit Testing.....	96
Cost Of The Bug.....	96
8. "if It Ain't Broke, Don't Fix It".....	98
9. Being Loved By The Company.....	99
10. Having "quality" And "happiness Of Users" As Fundamental Principles Of Company Philosophy.....	101
Programming & Bug Fixing.....	101
Syntax Bugs, UI Bugs, Logical Bugs.....	102
Code Freeze.....	107
Remember This For The Rest Of Your Testing Career.....	107
Test Case Reviews.....	110
Testing And Bug Fixes.....	111
Release.....	113
Major Release & Minor Release (Feature/Patch/Mixed).....	114
Emergency Bug Fixes And Emergency Feature Requests.....	116
Building ShareLane: Billy, Willy, Zorg And Star.....	119
ShareLane Architecture.....	120
Bug In Cc_transactions: File It NOW!.....	121
Using CVS.....	123
Builds.....	123
Releasing 1.0 To Production.....	125
Nightmare With 2.0: To Rollback Or Not To Rollback.....	127
CVS Branches And A Story Of Smiling Eddy.....	128
3 States Of CVS Branches.....	130
Bug Postmortems.....	132
Beta Releases.....	132
Nice Message;Releasing To 1 Machine;Release Moratorium.....	133
The Big Picture Of The Cycle.....	134
Lecture Recap.....	137
Questions & Exercises.....	142
Lecture 5. The Software Testing Life Cycle.....	144
Quick Intro.....	145
Testing New Vacuum Cleaner.....	145
Research.....	146
Documentation/Humans/Exploring.....	147
Test Planning.....	148
Test Execution.....	149
Test Education And Reality.....	150

2+2=4.....	150
Lecture Recap.....	150
Questions & Exercises.....	151
Lecture 6. Classifying the Most Common Types of Testing.....	152
Quick Intro.....	153
By Knowledge Of The Internals Of The Software.....	154
Black Box Testing.....	154
Scenarios And Patterns Of User Behavior.....	156
White Box Testing.....	158
Your First Bug Found With White Box Testing Approach.....	158
Use Cases Vs Test Cases.....	159
Understanding Test Coverage.....	160
Grey Box Testing.....	161
By The Object Of Testing.....	162
Functional Testing.....	162
Ui Testing.....	162
Difference Between UI Testing And Testing Using UI.....	162
Usability Testing.....	163
Localization Testing.....	163
Load/performance Testing.....	164
Security Testing.....	164
Compatibility Testing.....	164
By Time Of Test Execution.....	165
Before Any Release (alpha Testing).....	165
After A Beta Release (beta Testing).....	166
By Positivism Of Test Scenarios.....	166
Positive Testing.....	166
Negative Testing.....	166
Error Messages.....	167
Error-handling.....	167
By Degree Of Isolation Of Tested Components.....	168
Component Testing.....	168
Integration Testing.....	168
System (end-to-end) Testing.....	168
Test Preps For “5% Discount” Feature.....	168
Thoughts About Test Emails.....	175
By Degree Of Automation.....	176
Manual Testing.....	176
Semi-automated Testing.....	176
Automated Testing.....	177
By Preparedness.....	177
Formal/documenting Testing.....	177
Ad Hoc Testing.....	177
Lecture Recap.....	178
Questions & Exercises.....	181
Lecture 7. Test Preps.....	183
Quick Intro.....	184
The Tester's Mental Attitude.....	185

Logs, Landscapes And Material For An Article.....	185
Cycle Of Mistrust.....	186
Positive Testing And Destructive Thinking.....	186
Intro To Special Skills In Bug Finding.....	186
Black Box Testing Techniques.....	187
Dirty List/white List.....	187
Test Tables.....	189
Flowcharts.....	192
Risk Analysis.....	195
Hotel In The Mountains And “absolutely Obvious” Things.....	195
Example With Auto Parts.....	198
Equivalent Classes.....	201
Boundary Values.....	203
A-to-E.....	204
Testing Discounts On ShareLane.....	204
When You Start To Implement Testing Techniques.....	210
Be A Master Of Test Techniques, Not Their Slave.....	210
Lecture Recap.....	210
Questions & Exercises.....	211
Lecture 8. Bug Tracking.....	212
Quick Intro.....	213
The Bug Tracking System.....	213
Bug Attributes.....	215
ID.....	216
SUMMARY.....	216
DESCRIPTION.....	217
Basic Web Page Elements.....	218
Text.....	219
Link.....	219
Mailto.....	219
Anchor.....	219
404 - File Not Found.....	220
DNS Error - Cannot Find Server.....	220
Relative URL.....	220
Absolute URL.....	220
Image.....	221
Linked Image.....	221
Text Box.....	222
Captcha.....	222
Text Area.....	224
Password Input Box.....	224
Screenshots.....	224
Professional Ethics Regarding Passwords.....	224
Keystrokes.....	224
Drop-down Menu.....	225
Radio Button.....	226
Checkbox.....	227
Reset Button.....	227
ATTACHMENT.....	228
How To Make Graphic Attachments.....	228

SUBMITTED BY.....	229
DATE.....	229
ASSIGNED TO.....	229
The Concept Of The Bug Owner.....	229
Who Does What/Who Owns What.....	229
ASSIGNED BY.....	231
VERIFIER.....	231
COMPONENT.....	232
FOUND ON.....	233
VERSION.....	233
BUILD.....	233
DB.....	233
COMMENTS.....	233
SEVERITY.....	233
Bug Severity Definitions.....	234
PRIORITY.....	236
Bug Priority Definitions.....	237
Bug Resolution Times.....	238
Go/No-Go Criteria.....	239
BTW About Selling Bugs To Developers.....	239
ALSO NOTIFY.....	240
CHANGE HISTORY.....	241
TYPE.....	241
STATUS.....	241
RESOLUTION.....	242
Reported.....	243
Assigned.....	243
Fix In Progress.....	243
Fixed.....	244
Fix Is Verified.....	244
2 Steps Of Bug Regression.....	244
Verification Failed.....	245
Cannot Reproduce.....	245
4 Scientists And 1 Flask.....	246
Duplicate.....	248
Not A Bug.....	249
3rd Party Bug.....	250
No Longer Applicable.....	250
Bug Tracking Procedure.....	251
BTP Flowchart.....	251
Associations Between The BTS Attributes And Actions Taken To Resolve The Bug.....	252
Quick Closing Note About BTS And BTP.....	255
Effective Simplicity.....	255
Lecture Recap.....	255
Questions & Exercises.....	257
Lecture 9. Test Execution: New Feature Testing.....	258
Quick Intro.....	259
Smoke Test And Blocking Issues.....	259
QA Meeting About New Features.....	260
Test Estimates.....	261

Release Schedule And QA Schedule.....	261
1. What Is The Complexity Of The Feature To Be Tested?.....	262
2. Do You Have Any Experience With Testing Similar Features?.....	262
3. Is There Any Planned Integration With The Vendor's Software?.....	263
Entry/Exit Criteria.....	263
Test Plan.....	264
GENERAL INFO.....	265
INTRODUCTION.....	266
SCHEDULE.....	266
FEATURE DOCUMENTATION.....	266
TEST DOCUMENTATION.....	267
THINGS TO BE TESTED.....	267
THINGS NOT TO BE TESTED.....	267
ENTRY/EXIT CRITERIA.....	268
SUSPENSION/RESUMPTION CRITERIA.....	268
OTHER THINGS.....	268
Aggressive Testing From Jason Fisher.....	268
Lecture Recap.....	269
Questions & Exercises.....	270
Lecture 10. Test Execution: Regression Testing.....	272
Quick Intro.....	273
How To Choose Test Suites For Regression Testing.....	273
1st Group Of Test Suites For RT.....	274
2nd Group Of Test Suites For RT.....	274
Key-value Pairs.....	274
Test Case/suite Prioritization.....	275
How To Resolve The Contradiction Between Our Limited Resources And The Ever-Growing Number Of Test Suites.....	278
1. Prioritization Of Test Suites And Test Cases.....	278
2. Test Suite Optimization.....	278
3. New Hires Or Outsourcing.....	278
4. Test Automation.....	278
Automation Of Regression Testing: Do It Right OR Forget About It.....	278
A Story About The Merciless Automator, Benny M.....	279
What Exactly Was The Problem With ShareLane TA?.....	280
3 TA Components.....	281
TA Killer Called "MAINTENANCE".....	282
\$200 Shoes For A Baby.....	283
1. HELPERS.....	283
\$36,360 In Savings After 30 Minutes Of Work.....	285
2. AUTOMATION SCRIPTS.....	285
A. Tools For Component Automation.....	285
Concept Of The Length Of The Road.....	285
Short, Isolated Test Flows.....	286
B. Scripts For End-to-end Automation (E2E TA)	286
Money Savers And Money Eaters.....	287
4 IFs.....	288
When Test Automation Produces An Error It Means That.....	288
Test Automation And QA Team Morale.....	289

Questions To Ask BEFORE Starting To Write TA.....	289
A. How Stable Is The Tested Piece Of Software?.....	290
B. What Are The Frequency And Length Of Manual Execution Of The Candidate For Automation?.....	290
C. What Is The Priority Of The Test Suite/case To Be Automated?.....	290
D. How Much Longer Will That Task Be Around?.....	290
E. How Hard Will It Be To Write TA?.....	290
Two Important Factors About TA Programming.....	290
1. The Architecture Of TA.....	291
A. Ease Of Maintenance.....	291
B. Ease Of Expansion.....	292
C. Ease Of Code Reuse.....	292
2. The Technologies/tools Used For TA.....	293
Free Programming Languages Vs Commercial Tools (SilkTest, WinRunner And Others).....	293
8 Reasons Not To Use Commercial Tools.....	293
What If You Want To Specialize In SilkTest Or WinRunner.....	295
Idea About Test Automation With Wget And Python.....	296
When Regression Testing Stops.....	296
Lecture Recap.....	297
Questions & Exercises.....	299
Lecture 11. How to Find Your First Job in Testing.....	301
The Brick Walls Are There For A Reason.....	303
Why You Have A REAL Chance To Find A Job In Testing.....	303
Mental Tuning.....	304
What REALLY Prevents A Person From Getting An Entry-level Job In Testing.....	304
About Negativity And Skepticism.....	304
About Strong Desire To Make Good Money + A Complete Lack Of Any Desire To Work.....	304
Wrong Attitude/Right Attitude.....	304
"I'm Willing To Work Unlimited Hours."	305
"I'm Willing To Work On Weekends And Holidays."	305
"I'm Willing To Work For Any Amount Of Money."	305
Example With Housekeeper.....	305
Job Hunting Activities.....	306
ACTIVITY 0: Tune Up Your Attitude.....	306
ACTIVITY 1: Let People In Your Network Know.....	306
6 Degrees Of Separation.....	307
Using LinkedIn.....	307
ACTIVITY 2: Create A Resume.....	308
Resume As Presentation.....	308
STEP 1: List Of Achievements And Testing Related Experiences.....	309
Wendy And Her Print Shop.....	309
Done_stuff.txt.....	311
Action Verbs And Powerful Adjectives.....	311
STEP 2: Put Your White List Into Your Resume Template.....	311
Getting Beta Testing Experience.....	312
STEP 4: Populate The Section Labeled Software Experience.....	312
STEP 5: Populate All Other Sections Of The Resume.....	312
STEP 6: Polish The Language Of Your Resume.....	313
ACTIVITY 3: Working With Recruiters.....	314
Concept Of Target Market.....	314
The BIG Four.....	314

ACTIVITY 4: Launch A Campaign Dedicated To Self-promotion.....	316
Passive Search.....	318
Active Search.....	318
Shoot At ALL Targets.....	319
ACTIVITY 5: Interview Successfully And Get A Job.....	319
Homework Before The Interview.....	320
A. Get Info About The Company.....	320
B. Involve Your Network.....	320
C. If Possible, Try To Actually Use The Company's Product(s).....	321
D. Nice Haircut, Good Looking Clothes And Shoes, And Some Sleep.....	321
Phone Screening And Phone Interview.....	322
At The Day Of The Interview.....	323
1. Arrive On Time.....	323
2. Have A Firm Handshake And Look Directly Into The Interviewer's Eyes.....	323
3. Answer Questions Without Any Unnecessary Details.....	323
Meet The Parents, Monet And Sunflowers.....	323
4. Be Friendly, Yet Considerate.....	324
Anecdotes About Lovers Jumping From The Balcony, Etc.....	324
5. If The Interviewer Wants To Talk, Let Him Or Her Talk.....	324
6. NEVER Speak Negatively About Your Previous Or Current Employers.....	324
NEVER Be Negative At An Interview.....	325
"I Never Say Negative Things About My Employers.".....	325
Negativity Is An Opportunity Killer.....	325
7. Always Remember That During The Interview, The Interviewer Is Analyzing You As A Potential Coworker.....	325
Sharing The Passion.....	326
8. Honesty And Sincerity Win Hearts. Lies And Attempts To Conceal Something Are Sure Ways To Ruin Your Interview.....	326
"I Don't Know" – "If Needed, I'll Learn It" Combo.....	327
9. Don't Get Upset Or Angry If The Interview Doesn't Go Smoothly.....	327
"Can I Have Number 3, Please?".....	327
It Doesn't Matter What YOU Think About How Interview Is Going.....	328
The Best Techniques To Stay Calm During The Interview.....	328
10. Never Cancel An Interview Until You Accept A Job Offer.....	328
Interviewing Well Is A Separate Skill.....	329
11. Remember That An Interview Is A DIALOG, Not An Interrogation.....	329
"Death Or The Right Answer To What A Test Plan Is.".....	329
12. Use Professional Terms.....	330
13. Remember Your Mantra And Make Sure The Interviewer Knows These Things About You.....	330
14. List Of Typical Interview Questions And Recommended Answers.....	330
Q. Why Did You Decide To Become A Software Tester?.....	330
Q. What Do You Like Most About Software Testing?.....	330
Q. What Are The Key Qualities Of A Good Tester?.....	330
Q. Tell Me About Your Short- And Long-term Plans For Your Career In Software Testing.....	331
Q. In The Very Unlikely Scenario That The Company Needs You To Come Into The Office During The Night, Would You Be Willing To?.....	331
Q. In The Very Unlikely Scenario That The Company Asks You To Come In During Weekends And Holidays, Would You Be Willing To?.....	331
Q. During Crunch Time You Might Need To Work More Than Eight Hours A Day. Are You Comfortable With That?.....	331
Q. Can You Work On Several Projects At Once?.....	331

Q. How Do You Deal With Time Pressure And Pressure From The Management?.....	331
Q. Describe Your Relevant Experience And Education.....	332
Q. What Is Your Biggest Professional Achievement?.....	332
Q. Why Are You Leaving Your Current Employer?.....	332
Q. What Are Your Biggest Disappointments At Your Present Position?.....	332
Q. Would You Prefer To Work For A Large, Established Company Or A Start-up?.....	332
Q. Give Me An Example Of A Complex Situation And Solution That You've Found.....	332
Q. Tell Me About Your Strengths And Weaknesses.....	332
Q. What Would You Like To Improve In Your Career, And What Are You Doing About It?.....	333
Q. Would You Prefer To Work As A Member Of A Team Or Independently? Why?.....	333
Q. Why Do You Want To Work For Our Company?.....	333
Q. What Do You Know About Our Company? Have You Ever Used Our Product?.....	333
Q. Why Should We Hire You Over Another Candidate?.....	333
15. Make A Speech At The End Of Your Interview.....	333
16. Always Send A Thank-you Email To The Interviewer After The Interview.....	333
Jack London And 600 Rejection Letters.....	334
What Happens After They Interviewed You.....	335
Story About George And Olga.....	335
Lecture Recap.....	336
Questions & Exercises.....	338
Lecture 12. Bonus Tracks.....	339
How Much Technology Knowledge Must A Beginner Tester Have.....	340
"I Hear And I Forget. I See And I Remember. I Do And I Understand."	340
What To Do If You Are The First Test Engineer At A Software Start-Up.....	341
1. INSTALL A BUG TRACKING SYSTEM.....	341
2. LEARN THE PRODUCT, CREATE A SET OF ACCEPTANCE TESTS, AND START EXECUTING THEM BEFORE EACH RELEASE.....	342
3. BRING GOOD STANDARDS AND PROCESSES TO THE ATTENTION OF YOUR COLLEAGUES AND MANAGEMENT.....	343
What If You Are Asked To Do Test Automation As Your First Task.....	346
Performance Testing And Black Box Tester.....	346
What Are Stock Options.....	346
How People Get Rich With Stock Options. Example.....	347
Definition/Exercise Price/Vesting Schedule/Acceleration In Vesting.....	347
Millionaire On The Beach.....	349
How To Spot A Promising Start-up.....	349
Insane Popularity.....	350
When Start-up Is Being "pregnant".....	350
Why I LOVE Google, Inc.....	352
Always Being A Winner.....	352
Afterword.....	354
Links & Books.....	358
Glossary.....	360
Index.....	390

Introduction

**In theory, there is no difference between theory and practice.
But, in practice, there is.**
- Yogi Berra

If you don't have good dreams, Bagel, you got nightmares.
- Boogie (from the movie *Diner*)

Hello!

My name is Roman Savenkov. Welcome to my Practical Course, **How to Become a Software Tester**.

Quick info about myself:

- I'm a professional QA/Test engineer with **8+ years of Silicon Valley experience**.
- My professional accomplishments include **starting and building** test departments for several Silicon Valley start-ups, the most notable of which is **PayPal**.
- My teaching experience comes from tutoring in the areas of **software testing** and **Python**.
- In 2006, I published a book on software testing, *Testing Dot Com* (in Russian), which became a **best seller** on the largest Russian online book retailer, Ozon.ru. Many readers have found their first jobs in software testing after studying this book and following my recommendations about the job search.

After I received positive feedback from my Russian speaking audience (THANK YOU!), I decided to translate my book into English. Soon after I began the translation process, I realized that I could provide much greater value if I created not simply a book, but a **Practical Course** with written lectures accompanied by actual Web-based software.

So, I created this Course:

- **To give my readers (my students) practical skills.** There are a number of good, in-depth books on software testing. The problem lies in the HUGE gap between **reading about** something and **being able to do** something. Take kung fu, for example. Is it possible to become a kung fu master by simply reading about kicks and punches without kicking and punching? The same is true with software testing: to become a solid tester requires a lot of hands-on experience, and you cannot get that experience by reading without practicing. This Course was created to give you an opportunity to get that

experience...at your own pace...without paying the thousands of dollars usually needed for a practical education in testing.

- **To help my students find their first jobs in software testing.** My personal story is that I was a poor immigrant who came to the U.S. for a better life. I didn't have a Plan B (like landing at my parent's apartment) if something "didn't work out." I had only one plan: to succeed. My situation required focused effort to obtain a job, and to do so I needed **solid, targeted, practical** skills in two areas:

- 1. How to test Web-based software**
- 2. How to make money as a software tester**

That's why this Course consists of a set of how-to lectures and practical exercises designed to help you learn:

- > **Software testing skills**
- > **Job hunting skills**
- > **Job keeping skills**

I was lucky to have two amazing tutors: **Alex Khatilov** and **Nikita Toulinov**. It was their valuable experience and kindness that helped me to get started with my career. Now it's my turn to help YOU!

Why is this Course different?

- This Course is a set of lectures **in the form of a book**, but here is the thing: the material in these lectures is linked to an actual software project, www.ShareLane.com. ShareLane is a test application that I created to:

- > Illustrate the examples
- > Enable you to interact with real software as you go through material
- > Enable you to look into the software code so you can see the root causes of software bugs
- > Enable you to view the contents of the database and log files so you can see the relationship between user activities and the Web site back end
- > Enable you to use test automation and see how it was written.
- > Enable you to file bugs into the bug tracking system.

BTW

When you see this sign: **SL**, it means that you should go to ShareLane.com and interact with its software or use its materials. I strongly encourage you to practice and learn as much as possible using the resources you'll find there.

- This Course is about action – **NO DRY THEORY** here. I'm going to give you the approaches, methods, ideas, and **brain positioning** that have **immediate practical application**. Under "**Brain Positioning**," I explain the most vital fundamental concepts and attitudes regarding the subject.

- We'll cover many topics that are not usually found in software testing tutorials: e.g., how to prepare for and successfully pass interviews; practical advice for an inexperienced tester who joins a start-up (young company) and has no idea how to begin; nuances about office politics, etc.
- The testing approaches that I recommend have been **successfully implemented at PayPal**, one of the top software companies in the world. You'll learn concrete practices that have been used to test:
 - > Very complex financial software
 - > Money-transferring service where a tester's mistake can cost millions of dollars in damages

Who is the primary audience for this Course?

- Anyone who wants to get a job as a software tester
- Black box testers who want to expand their knowledge of grey box testing

Who else will find this Course useful and interesting?

- Anyone involved in software development
- Recruiters working with/for software companies
- Those who want to learn about the inner workings of software companies

Two important points:

1. On the one hand, testing activities are not directly regulated by any kind of formal laws (e.g., the way a stockbroker's activities are). On the other hand, in our industry, methodologically nothing is set in stone.

Therefore, the whole business of building an effective system for the search and prevention of software bugs is entirely up to the professionalism, creativity, and attitudes of concrete testers and QA engineers working for a concrete company!

That's why many things that we'll be covering (approaches, documentation, processes, even terms):

- Have large number of variations in existing software companies
- Can be used in the suggested form, or – even better – be adjusted for the concrete company - where you work now (or will be working in the near future).

2. What works for one environment doesn't necessarily work for another. So:

- test approaches
- complexity of processes
- volume and types of documentation

that effectively work for large companies are not necessarily acceptable in start-ups, and vice versa.

We'll focus on how to test in the Internet start-ups with the aim of creating a foundation for the QA department of a large software company.

We're almost finished with the Introduction. I want to mention several other things before we get started.

A note about Test Portal

Test Portal is a special site for ShareLane testers (i.e., for you!). IMHO, it's the most important part of ShareLane. Test Portal gives you an opportunity to do a variety of things: use test automation, look into the database and software code, file bugs, and so on. **The link to Test Portal is located at the bottom of the ShareLane pages.**

Let's agree that we will use the ">" (more than) sign to separate the steps of a path when using Test Portal. For example, Test Portal>DB>Data>users means that first you click the "Test Portal" link, next you find the "DB" section, then you click the "data" link, and then you click the "users" link.

I encourage you to use Test Portal as much as possible! This is your real chance to learn software testing on an actual software project.

My contact info

My personal email is roman@qatutor.com. **I'll be happy to hear from you!** Being a self-published, independent author, I fully depend on your recommendations, so if you like this Course don't be shy about sharing your positive feedback with your friends, relatives, coworkers, jogging partners, and everybody you'll meet for rest of your life.

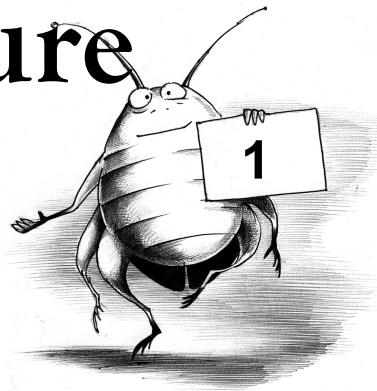
Let's begin our journey!

Disclaimer

1. This Course is not intended for those who are planning to test critical software: e.g., software used for the monitoring of heart muscle contracts or software used for military purposes.
2. Any resemblance to events, locales, or persons, living or dead, is entirely coincidental.
3. Product names, logos, brands, and other trademarks featured or referred to within this Course (including qatutor.com and sharelane.com materials) are the property of their respective trademark holders. These trademark holders are not affiliated with this Course. They do not sponsor or endorse this Course.
4. The information provided in this Course is without warranty of any kind and should not be interpreted as legal advice. Owners of ShareLane.com and QATutor.com will not be liable for any direct, indirect, or consequential loss or damage arising in connection with their Web sites, whether arising in tort, contract, or otherwise, including, without limitation, any loss of profit, contracts, business, goodwill, data, income, revenue, or anticipated savings.

**To my mother and my wife
I love you**

Lecture



A Bug or Not a Bug

Lecture 1. A Bug Or Not A Bug.....	9
Quick Intro.....	10
Three Conditions Of A Bug's Existence.....	11
The Gist Of Testing.....	11
Spec, Spec, Spec.....	12
Software Bugs And Spec Bugs.....	13
Other Sources Of Expected Results.....	14
Why We Call Them "Bugs"?.....	17
Lecture Recap.....	17
Questions & Exercises.....	18

All truths are easy to understand once they are discovered;
the point is to discover them.
- Galileo Galilei

In the fields of observation chance favors only the prepared mind.
- Louis Pasteur

Quick Intro

According to the logical law of the excluded middle, each item is **either P or not-P**. There are no other alternatives; i.e., if you have Rolex watch #F787999, any object in universe will be:

- **EITHER** your Rolex #F787999
- **OR** not

Imagine that you stand at the end of an assembly line where certain objects come toward you one by one. Your task is simple: to **expect** the appearance of your Rolex #F787999 and confidently call out "**BUG!**" if **anything else** shows up.

So:

- a pack of buttermilk
- an electronic alarm clock
- Rolex #F787998

anything else except our Rolex #F787999

will be labeled a "**bug**."

Now, let's find similarities in the following three situations:

1. Your new girlfriend or boyfriend claims that she or he is a great chef and in the morning she or he cannot even make a piece of toast.
2. You are reading a book on software testing and it tells you about toast making.
3. Your girlfriend or boyfriend from #1 has read the book from #2, but the toast is burnt to a crisp.

All these situations are similar because each of them has a **mismatch between the actual result and the expected result**.

1.

Expected result: *girlfriend or boyfriend knows how to cook*
Actual result: *morning without toast*

2.

Expected result: *talk about software testing*
Actual result: *talk about girlfriends, boyfriends, and making toast in the morning*

3.

Expected result: *toast is finally made*
Actual result: *another toastless morning*

Here is the definition of a “bug”:

A bug is a deviation of an actual result from an expected result.

According to the logical law of the excluded middle, **we have a bug in ANY case where an actual result deviates from an expected result.**

Three Conditions Of A Bug's Existence

A bug exists only if ALL of the following three conditions are met:

- 1. We know the expected result**
- 2. We know the actual result**
- 3. We know that the actual result deviates from the expected result**

BTW

My advice: Every time you see a mismatch between the actual and expected, label that situation with the word "bug." As time goes on, you'll develop a habit, which will gradually grow into a reflex. Please note that for the sake of this training, it doesn't matter how futile, cheap, or temporary your expectations are - the main thing here is to gain the skill of **automatic bug recognition**.

Example

Here are some more bugs from real life:

1. A nice face backed by an ugly personality.
2. Parrots say the nastiest words at absolutely inappropriate times.
3. Hollywood story: reach fame and drink yourself to rehab.

The Gist Of Testing

The gist of any testing is a search for bugs. Whether we try a new food processor, observe the behavior of a girlfriend or boyfriend, or torture ourselves with self-analysis - we are looking for bugs.

Here is how bugs are found:

- 1. We find out (or already know) the expected result.**
- 2. We find out (or already know) the actual result.**
- 3. We compare the actual result with the expected result.**

As you can see, each of us is an experienced tester ALREADY, because bug finding is an essential part of our existence.

Example

As an illustration of the correct approach, I can tell you about my pal who has developed a whole system of argumentation to support his thesis that humans and computers are built on the same principle. The cornerstone of his theory is the fact that both humans and computers have:

- physical containers (body/hardware)
- intangible content (soul/software)

Thus, when he gets sick he says "I've got a hardware problem," and when he does stupid things he says, "My software is buggy."

Now, let's remember that we are here to learn about software testing. It's more or less clear how to get an actual result - we just have to observe how the software behaves. Obtaining the expected result can be far more complex.

Spec, Spec, Spec

The main sources of expected results are:

1. Specification
2. Specification
3. Specification
4. Life experience, common sense, communication, standards, statistics, valuable opinions, etc.

Specification being stated three times in a row is not a problem with my keyboard, but my way of stressing that specification is:

- the most important
- the most valuable
- the most respected

source of expected results within your software company. Specification is as important for a tester as a script is for an actor or traffic laws for a driver. So, what is **specification**?

Specification (or "spec" for short) is a detailed description of how software should work and/or look.

Please note that some startups just don't have specs - in a minute we'll discuss what to do in that case.

Example

Item 2.1. of the spec #1111 "New User Registration" states:

"Text field 'ZIP code' is required. Page with error message should be displayed if user submits registration form without 'ZIP code' filled."

SL

Testing is simple and you can do it yourself:

1. Go to www.sharelane.com.
2. Click link "Sign up".
3. Press button "Continue".

If an error message is NOT displayed, we have to report a bug.

If error message is displayed, we can be confident for a while that there is no bug here.

Why did I say "for a while"? We'll talk about it during our conversation about regression testing.

Software Bugs And Spec Bugs

Let's assume that the error message wasn't displayed. In this case, we have a **classic case of a software bug; i.e., a bug born because of a mismatch between the actual behavior of the software and the expected behavior (which is usually specified in the spec)**.

If you paid a little bit of attention when you read item 2.1., you surely noticed (joke) that it wasn't clear what kind of error message is expected to be displayed—i.e., the decision on the actual text of error message is up to the programmer, and he or she can write code that will produce:

- a NONinformative, irritating message: "**Error**", leaving the user to wonder what he or she has done wrong and cultivating a feeling of guilt in him or her, OR
- an easy-to-understand message: "**Oops, error on page. ZIP code should have 5 digits**".

Formally, the programmer will be right in both cases, because the specification doesn't elaborate about the text of the error message.

BTW

In early days of start-up when software changes very often, there is no need to include precise text of error messages into the spec. The **idea** about error message will do. For example: "Error message should state that ZIP code should consist of 5 digits".

Here we have a **situation where the spec itself is buggy**, because

- we reasonably **expect** that spec to give us details about the error message, and
- the **actual** spec doesn't give us those details.

Let's call this situation "**spec bug**".

Brain positioning

Each found bug (both in software and in specs) must be filed into the bug tracking system. There should be no exceptions. Bug reporting is one of the most important parts of our profession.

We'll have a separate lecture to talk about bug reporting and tracking.

Brain positioning

From the start, you must understand one very important thing: **when you file a bug against someone, it doesn't mean that you hate or disrespect that person.** It simply means: "I have found mismatch between actual and expected. Please, fix the problem."

If somebody takes a bug personally, you must have the respect and patience to explain to him or her that there is nothing personal here - this is just a part of your job.

Other Sources Of Expected Results

Now, let's look at some other sources of expected results, in addition to the specs:

- Life experience
- Common sense
- Communication
- Standards
- Statistics
- Valuable opinion
- Other sources

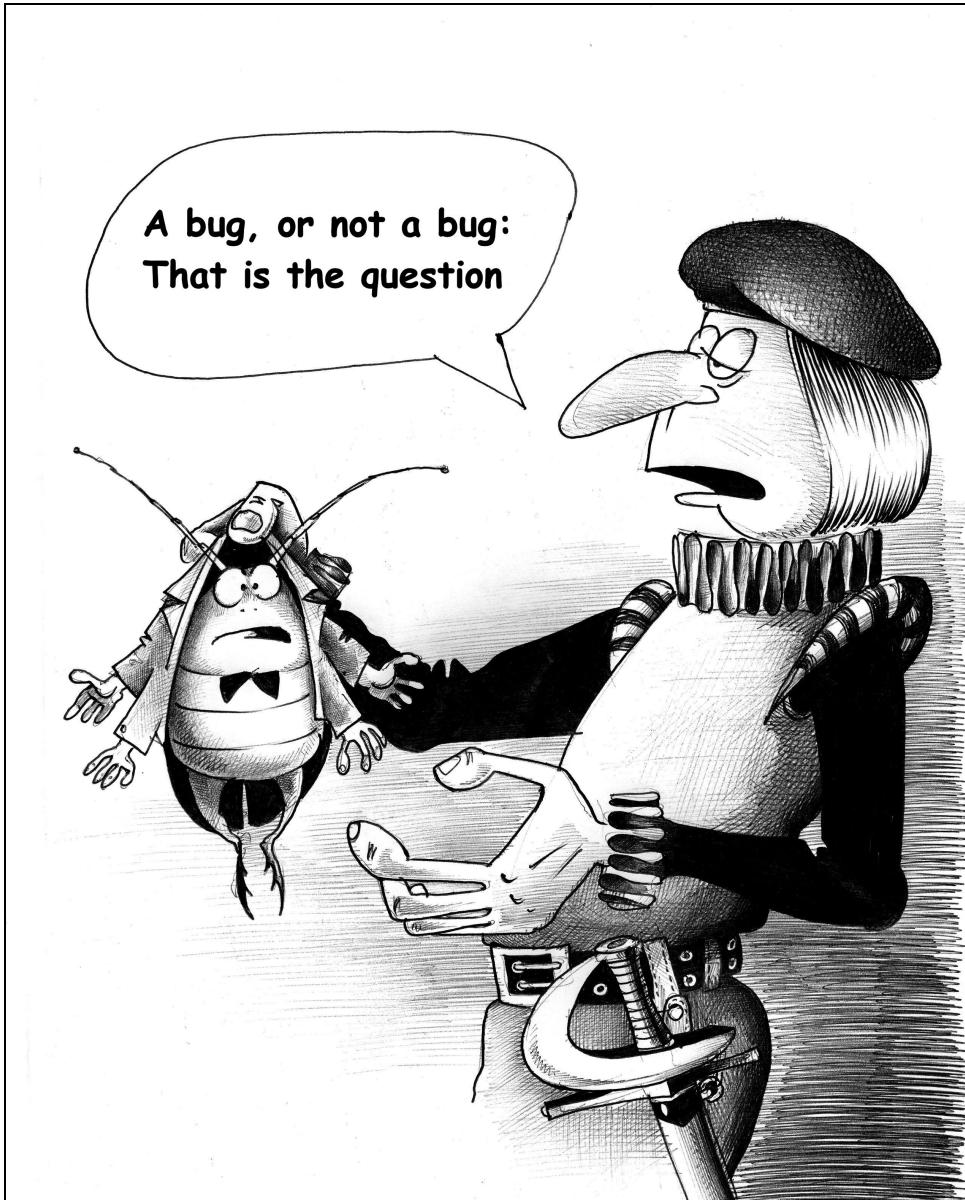
LIFE EXPERIENCE

Whatever we learn browsing the Web, reading books, working, studying, communicating, etc. is accumulated into the precious container called "life experience." Therefore, when there is no spec, we can dig up something from our life experience and give a reason why the expected result should be such and such.

Example

When there is no spec and the programmer enters text "Error" instead of an informative error message, you can file a bug against him or her, saying that error messages must give details to users about what was wrong and/or what should be done. Why? Because in your experience, it's very confusing to see only "Error" and have no clue about what you did wrong.

Of course, the programmer can say: "In my opinion, 'Error' is enough," and here the argument begins...The best way to prevent the argument is to have a spec in the first place. Spec that would give clear, concrete instructions, so everyone knows exactly what should be expected.



COMMON SENSE

This is one of our main allies, even if we have a spec.

Example

Let's say you are testing a Web site that enables users to upload their digital photos. The spec says that a user can only upload one photo at a time. But what if our user has 200 photos? Will he or she be happy if we require him or her to go through the boring process of uploading 200 times? I don't think so! What shall we do? Let's write an email to product_managers@sharelane.com and ask them about the bulk upload of pictures. We can also file a bug report with this request. This type of bug is called a "Feature Request."

COMMUNICATION

Even the best spec in the world has room for elaboration. There are also plenty of situations when there is no spec at all. What to do in both cases? **Communicate!!!** Talk to your colleagues. Don't be afraid to look dumb or to bother busy people. You, as a test engineer, ARE an important person, and you MUST get ALL the info you need.

Of course, you should always act in a discreet way, so if a person is truly overwhelmed with work, give him or her some time to get back to you.

BTW

Each job description for a tester position has a requirement for "excellent communication skills," and this is absolutely a vital thing for a tester. I myself have a strong foreign accent, English is my second language, and I realize that I have to put forth extra effort to make myself clear.

"Excellent communication skills" has two main components:

- a. How good you are with people
- b. How clearly you communicate

In order to excel in both ways, **you have to stop worrying and just try to do your best.** Communication is an essential part of our profession, and needed skills can be learned and mastered.

STANDARDS

Here are a couple of examples for illustration.

Example

The industry standard is that a user must receive an email confirming his or her registration at your Web site. So, you can reasonably expect that your Web site would follow this standard.

Example

Your company might have an internal standard that error messages must be in this format:

- Font "Times New Roman"
- Font Color "#FF0000" (color code for "red")
- Font size "110%"

so each parameter above ("Times New Roman," "#FF0000," and "110%") can serve as an expected result.

STATISTICS

"Four seconds is the maximum length of time an average online shopper will wait for a Web page to load before potentially abandoning a retail site" (<http://www.akamai.com>). This data can be used for filing bugs about the performance of the Web site.

Always remember that your customer is just one click away from your competitor.

VALUABLE OPINION

This can be the opinion of your boss.

OTHER SOURCES

There can be some other sources depending on your concrete project; e.g., if you test photo sharing Web site (like photobucket.com), you might need to read an article from the magazine for professional photographers.

Why We Call Them "Bugs"?

The following quotation is taken from a Wikipedia.org article:

"The first 'documented' computer bug was a moth found trapped between points at Relay #70, Panel F, of the Mark II Aiken Relay Calculator while it was being tested at Harvard University, 9 September 1947. Grace Hopper affixed the moth to the computer log, with the entry: '**First actual case of bug being found.**'"

Lecture Recap

1. A bug is a deviation of an actual result from an expected result.
2. Here is how bugs are found:
 - a. We find out (or already know) the expected result.
 - b. We find out (or already know) an actual result.
 - c. We compare the actual result with the expected result.
3. The main resource of expected results in software companies is the specification.
4. Specs might have problems themselves and sometimes there aren't any specs at all. So, other sources of expected results are
 - Life experience
 - Common sense
 - Communication
 - Standards
 - Statistics
 - Valuable opinion

- Other sources

Questions & Exercises

1. What is a bug?

2. Look for bugs everywhere. Include this word in your vocabulary and write down your most colorful bugs in this format:

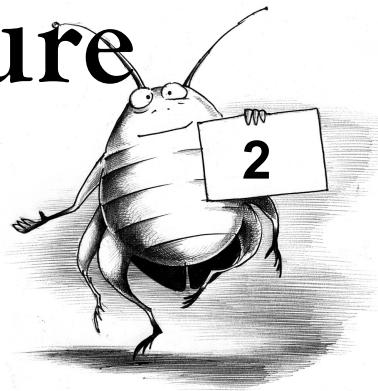
actual result = <short description of an actual result>

expected result = <short description of an expected result>

3. Think about other possible sources of expected results.

SL 4. Create new user account on ShareLane.com.

Lecture



The Purpose of Testing

Lecture 2. The Purpose Of Testing.....	19
Quick Intro.....	20
Exposing Misconceptions.....	20
How To Make Use Of Stats For Post-Release Bugs.....	26
Testing And QA.....	28
Lecture Recap.....	29
Questions & Exercises.....	29

Quality is not an act, it is a habit.
- Aristotle

First we make our habits, then our habits make us
- Charles C. Noble

Quick Intro

The purpose of testing is to *find and address* bugs BEFORE those bugs are *found and addressed* by users.

Let's elaborate on the terms "**find and address**."

Testers find bugs by using testing techniques.

Testers address bugs by:

- bug reporting and,
- after the concrete bug is fixed,
 - > verifying that the bug was *really* fixed
 - > checking to see if new bugs have been introduced as a result of the bug fix

Users find bugs by using our Web site.

Users address bugs by being pissed off and/or calling customer support, and/or going to our competitors, and/or filing lawsuits against us.

Every time a user finds a bug, it negatively affects his or her happiness in being our customer. We, as testers, care about the happiness of our users, and we show our care by effective testing.

Please note that if a tester finds a bug, but doesn't address it, his or her testing makes no sense, because an unreported bug will not be fixed. It's like having a winning lottery ticket and not taking it to the lottery organizers.

The ideal situation is when the testing process discovers and addresses ALL bugs. But the truth is that ideal situations are often far from reality when we are dealing with software. Read on!

Exposing Misconceptions

There are 2 misconceptions associated with the purpose of testing which you should be aware of, because they are widespread and harmful.

Misconception #1: "Testers must make sure that 100% of the software works fine."

Here is Spec #1522:

- 1.0. Program froggy.py accepts user input.
- 1.1. Text of prompt for input: "What animal do you like more: frog or cat?"
- 1.2. If input is either "frog" or "cat", the program must print on screen: "<user input> is a great animal".

- 1.3. If user enters nothing and just presses "Enter" the program should print message: "You didn't type anything".
- 1.4. In all other cases the message should be "You didn't enter the expected word".

Here is a code of froggy.py written in Python (the text after # is a comment for the current line of code):

```
user_input = raw_input("What animal do you like more: frog or cat?") #Display a prompt for text input.

animal_list = ['frog','cat'] #this is a list of 2 words one of which is expected to be entered

if user_input in animal_list: #if user entered word that matches any element inside animal_list

    print user_input + " is a great animal"

elif user_input == "": #if user entered nothing and just pressed Enter

    print "You didn't type anything"

else: #in all other cases

    print "You didn't enter the expected word"
```

Let's list the four possible inputs:

1. "frog"
2. "cat"
3. "" (Null input)
4. Any input different from input 1 to 3 inclusively

Brain Positioning

Input can mean different things depending on the context. We'll use an incremental approach for a smooth introduction to this and many other concepts. For now, **input is data passed to a system**.

From a user's perspective, the typical inputs are:

- Text, e.g., username entered during log in.
- File, e.g., image file uploaded to a photo-sharing Web site.

Brain Positioning

There are two kinds of input:

- Valid input
- Invalid input

We need to know the validity of the input because software must process valid and invalid inputs differently. In the above example, valid input can be determined by looking at the user prompt: "What animal do you like more: frog or cat?" The valid inputs are "frog" and "cat." All other inputs are invalid.

If the spec doesn't clarify which input is valid and which input is invalid, we can send an email to the product manager and/or file a bug against the spec. If there is no spec, we can use other sources for the expected result to find out about the validity of various inputs.

Depending on the situation, Null input can belong either to valid or invalid input.

In order to verify that 100% of the software "works," we have to create four inputs and verify four actual results against four expected results.

Test Case #1:

Input: "frog"

Expected Result: frog is a great animal.

Test Case #2:

Input: "cat"

Expected Result: cat is a great animal.

Test Case #3:

Input: <nothing>

Expected Result: You didn't type anything.

Test Case #4:

Input: "dragon"

Expected Result: You didn't enter the expected word.

Wait a minute! Why are we so sure that "dragon" covers all variety of possibilities called "**Any input different from input 1 to 3 inclusively**"?

Our confidence comes from fact that looking at the code of the extremely simple program `froggy.py`, we saw that **all** inputs (except "frog," "cat," or null) will be redirected by "else" to the output "You didn't enter expected word." But, as a rule, testers don't look into the software code at all! Why? Because:

- On the one hand, the programmer and tester usually receive the spec at the same time, so when the tester starts writing test cases, there is no code to look into.
- On the other hand, software usually has thousands of lines of code, and even programmers cannot be sure how it will *actually* handle each possible input.

So, let's assume that we don't look into the code. How can we verify that **100%** of the program `froggy.py` "works fine"? **We have to test all possible inputs in addition to inputs from Test Cases 1-3 inclusively.**

BTW

If the language of our input is English, we can type

94 printable characters:

- Letters (a-z, A-Z)
- Numbers (0-9)
- Special characters (punctuation and symbols; e.g., ";" and "\$")

and 1 invisible graphic character

- Space (produced by the space bar on a keyboard)

There are also 33 other ASCII characters*, most of which are obsolete. Let's not take them into consideration for the sake of example below.

*"American Standard Code for Information Interchange (ASCII), is a character encoding based on the English alphabet." (Definition from Wikipedia)

If we test the condition "**Any input different from input 1 to 3 inclusively**" with one character, we have 95 possible inputs. But what if a user enters two characters? In that case we have 9,025 possible inputs (95x95). What about three characters? We'll have 857,375 possible inputs; i.e., **857,375 test cases**. How about ten characters? (That would be 59,873,693,923,837,890,625 possible inputs). So, even in the case of a trivial task to test user input that's different from

- "frog"
- "cat"
- ""

there is a humanly unthinkable number of possible inputs! But the real pain only starts here.

The basic concept of the computer program is: **Input -> Processing ->Output**.

We can look at this analogy:

*When somebody calls your cell phone, it's an Input ->
When you look at the caller ID and decide to take the call or not, this is Processing ->
And then, depending on your decision, you either pick up or you don't, it's an Output.*

In order to test "Processing," certain Input must be fed into the program, so we can verify the Output. To produce Output, Processing takes certain logical routes or paths (i.e., "If this is my wife, I'll pick up. Otherwise I won't."). In our simple case, there are 3 paths:

Path 1: If the Input is "frog" or "cat", Processing produces the Output "**<animal> is a great animal**".

Path 2: If the Input is "", Processing produces the Output "**You didn't type anything**".

Path 3: If the Input is not "frog", "cat", or "", Processing produces the Output "**You didn't enter the expected word**".

In case of the worthless program froggy.py, each path is short, simple, and straightforward. In the case of real software, e.g.; software known as the "Google search engine", Processing is much more complex.

Example

Imagine that you are hiking in the park. You're walking along the trail, and you see a fork with two trails: "Montebello" on the left and "Skyline" on the right. You take the trail called "Skyline" and keep hiking. Half a mile later, you see another fork and make another choice of where to go, and so on. Now, if you took the trail called "Montebello," it would have been a different route for you. To explore the whole park, you'd have to hike all the trails.

Example

Now imagine the same concept, but on a larger scale. Imagine that you have to drive from San Francisco to New York City. I'm not talking about the shortest route; I'm talking about *all* possible routes. The combination of roads is probably not calculable; e.g., you can drive to Florida, then go back to California, drive to Michigan, drive to Minnesota, drive back to California again, and only after that, drive to New York City. The programmer friend of mine who gave me that example said, "**You know, it took two hundred years to build all those roads, but I can spend 20 seconds programming and get the same insane number of paths inside my program.**"

So, in the case of real software, processing can contain tens, hundreds, thousands, or even millions of unique, logical paths that must be tested if you want to test 100% of the software. What's important here is:

- In the case of inputs, you usually can determine a piece of data that represents some class of values (e.g., "dragon" represents the class "**Any input different from input 1 to 3 inclusively**"), so if the processing of the input "dragon" produces the expected result, you can assume (without being 100% sure!) that the input "*^98gB&T*" would produce the same result.

- In the case of logical paths, you usually cannot say that Path 1 works if Path 2 works! In order to say that Path 1 works, you have to **actually** test that path.

Therefore, on top of an enormous number of possible inputs, we have an enormous number of possible logical paths.

Let's pour some more salt on the wound. In addition to inputs and logical paths, we have *conditions* - i.e., presets and/or environments that exist when the software is used (you can also say "under which the software is used"). For example, execution of the same test case can end up with different results if you execute it using different Web browsers, e.g., Internet Explorer 6.0 and Firefox 2.0 (first condition is: "Test case is executed using IE 6.0"; second condition is "Test case is executed using Firefox 2.0"). So, in order to test **100%** of software, not only we have to test all inputs and all logical paths, but we also must execute them in **each version of each Web browser** known to humans.

Brain Positioning

Looking at the enormous mountain of possibilities produced by combination of

- inputs,
- logical paths, and
- conditions,

we can clearly see that **it's usually absolutely impossible to test 100% of the software**. In other words, **testing usually cannot cover 100% of the possibilities of how software can operate.**

Therefore, we can

- EITHER set up a 200-year plan to test software that probably nobody will need in 10 years.
- OR be smart testers.

Brain Positioning

Whether we like it or not, **there is always a probability that bugs will be missed by testers**. But we, software testers, can substantially reduce that probability by working hard and, most importantly, smart to find and address bugs.

Misconception 2: "A tester's effectiveness is reflected in the number of bugs found **before** the software is released to users."

First, let's introduce a new term: "**release**". Depending on the context, a release can be defined as:

- Either the process of passing on some version of the product* to the consumer
- Or some version of the product

* *In software terms, "product" is another term for "software."*

Here's what happens: First we develop and test software, and then we release it (make it available) to our users.

During this process, bugs can be found:

- Before a release – these are called pre-release bugs
- After a release – these are called post-release bugs

Pre-release bugs are found and addressed by folks inside the company (testers, developers, etc.). Post-release bugs are found and addressed by anyone using the software (e.g., a Web site), but mostly by our users.

Here is the question: Do users care how many pre-release bugs the testers found during testing? Let's see. We all use the Google search. Do we, as users, care how many bugs have been found by the Google testers? Nope. The only thing we care about is whether Google search retrieves relevant results. Users are happy when the software works fine, and they are not happy if it doesn't (because of bugs, servers being down, or whatever reason). **So there is no connection between user happiness (or more formally "customer satisfaction") and internal company data about how many bugs have been found.**

Now let's look at this from a software development perspective. Every release is different. It's different because of the complexity of the development and testing, the number of lines of code, the time given to develop and test it, and many other factors. Thus, **not all releases are created equal, and the number of findable bugs can vary substantially from release to release.**

Therefore, **the number of pre-release bugs means nothing and tester's effectiveness cannot be judged by it.** It's just a number.

What really matters is the number of post-release bugs and the seriousness of those bugs. That's why once post-release bugs emerge, we have to analyze what, if any, was done incorrectly, and what can be done to prevent releasing similar bugs in a future. Read on.

How To Make Use Of Stats For Post-Release Bugs

BTW

Each bug has a priority which reflects that bug's importance for business of the company.

Priorities usually range from P1 to P4, with P1 being the most critical: if user cannot register, it's P1; if some link has navy color instead of blue color, it's probably P4.

Here is the idea: **After each release, post-release bugs are analyzed for the purpose of finding weak link(s) in the software development process.**

(see next page)

Example

After the last release, we retrieved stats based on two parameters:

- functionality
- priority

Priority	Functionality	Registration	Search	Shopping cart	Checkout
P1		1	0	0	7
P2		0	1	0	2
P3		2	0	0	0
P4		3	2	4	0

As we can see, Checkout has the worst stats: 7 P1 bugs.

Now, let's assume that after each release we have a problem with Checkout, and we want to find out what is going on.

- All Checkout specs are written by a product manager, Alan.
- All Checkout code is developed by a programmer, Boris.
- Checkout was always tested by a tester, Connor.

The first thing we would probably do is kick the tester's ass, because he missed those bugs, but if we look deeper we'll find out that:

- Alan's specs are poorly written.
- Connor has married Boris's former fiancée and does his best to avoid him.
- Both Connor and Boris dislike Alan, because Alan is the company president's nephew, and he got hired without an interview.

After we have spent some more time, we'll also discover that:

- Alan has neither the background nor the documentation to fully understand all the nuances about Checkout, such as the integration with an electronic payment system.
- Boris and Connor are great professionals who perform above and beyond on all projects where they don't have to interact with each other.

Here we have a real investigation! Sometimes it takes rearranging people and projects to **prevent problems**. Doing this kind of analysis is not testing, but QA (Quality Assurance).

Testing And QA

Example

A certain Mr. G. realized one day that his life was boring and uneventful, so he left his wife and their little son. In order to spice up his existence, he moved to another city and started all over. Many years passed. One day his ex-wife calls him and asks him to influence his twelve-year-old son who has started to smoke, has been caught at the movies ditching math class, and has made several futile yet confident attempts to kiss his neighbor Jessica. The situation appears to require a father's firm hand.

The QA approach would be if ex-husband never left his wife and raised their son, trying to **prevent** bad seeds in son's behavior.

The testing approach occurs when the ex-husband

- compiles a list of issues like: "actual: movie; expected: math"
- locks the little rascal in a mountain cabin, limiting his spiritual and sexual life to French literature of XIX century
- hires private tutors who are supposed to look at the list and mold the boy to make sure that all the issues on the list are dealt with

Brain positioning

The purpose of QA is to prevent bugs

The purpose of testing is to find and address bugs that have not been prevented

QA and testing are **similar** because:

- Their common mission is to improve software

QA and testing are **different** because:

- QA improves software through **process improvement**
- Testing improves software through **bug reduction**

Although our Course is focused on testing, we'll also look at many issues from the QA angle.

Why is process improvement so important? Because software development IS a process (more precisely: *a set of processes*), and thus software quality has its roots in two things:

1. How quality-oriented our software development process is
2. The mentality and actions of **each** participant in the software development process*

* *The software development process is also called the software development life cycle – we'll spend a lot of time talking about this.*

This brings us to the important idea that **testers and QA engineers alone cannot be responsible for software quality**. You can be held responsible only for things that you FULLY control.

For example, if a spec has a bug, that bug can be incorporated into the test cases and the software code as a legitimate thing.

But what is so special about testers and QA engineers, then? **Our mission is to be passionate advocates of constant quality improvement in any shape and form.** While we cannot be in full control of software quality, it's in our hands to go above and beyond promoting quality as a vital philosophy within a software company.

BTW

In theory, QA engineers and testers should do different things, but in reality, QA engineers are usually testers who know how to improve various processes.

It's also a common business practice to use "QA engineer" and "test engineer" (or shortly "QA" and "tester") interchangeably. This same business practice is used with the terms "QA department" and "test department": both names are legitimate.

BTW

It's much easier to do process improvement in start-ups than in big companies, because start-ups have

- More open environments
- More flexibility in decision making
- Less red tape
- Less people

Lecture Recap

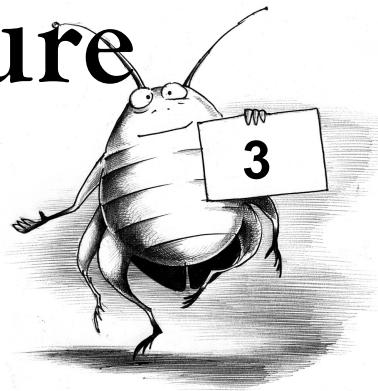
1. The purpose of testing is to find and address bugs before those bugs are found and addressed by users.
2. Testing usually cannot cover 100% of the possibilities of how software can operate.
3. The number of pre-release bugs doesn't matter.
4. An analysis of stats for post-release bugs can help identify weak links in software development.
5. QA is targeted towards prevention.
6. Quality is a group effort. Each person involved in software development is responsible for quality.
7. Quality is directly affected by the effectiveness of the software development process.

Questions & Exercises

1. What is the purpose of testing?
2. How do testers find and address bugs?
3. How do users find and address bugs?
4. Why is it usually not possible to test 100% of software?
5. How many printable characters do we have if the language of input is English? Name three groups of those characters.
6. Why number of found pre-release bugs has no meaning?
7. What would you do as a manager if you knew what really caused the problems with Checkout?
8. How testing contributes to the software quality?
9. How QA contributes to the software quality?
10. List some more analogies that demonstrate the difference between QA and testing.



Lecture



Test Cases and Test Suites

Lecture 3. Test Cases And Test Suites.....	31
Quick Intro.....	32
Test Case Structure.....	32
Results Of The Test Case Execution.....	34
Useful Attributes Of The Test Case.....	35
Data-Driven Test Cases.....	38
Maintainability Of Test Cases.....	38
The Number Of Expected Results Inside One Test Case.....	42
Bad Test Case Practices.....	45
Test Suites.....	51
States Of A Test Case.....	59
The Importance Of Creativity.....	60
Three Important Factors That Affect Test Cases.....	61
Checklists.....	62
Lecture Recap.....	63
Questions & Exercises.....	64

**I am not discouraged,
because every wrong attempt discarded
is another step forward**
- Thomas Edison

Do, or do not. There is no 'try'
- Yoda ('The Empire Strikes Back')

Quick Intro

Before going on a fishing trip, Mr. Wilson came up with the following list:

1. Fishing pole
2. Box with extra hooks, fishing lines, and other gear
3. Can with worms
4. Huge beer mug
5. Large bottle of Sapporo
6. Potato chips

In the morning, trying not to wake up his wife, kids, two cats and the canary, Mr. Wilson takes that list and checks his backpack for the availability of each of the 6 objects.

Each item on his list is a **test case**.

The list itself is a **test suite**.

The process of figuring out and writing down each item is **test case generation**.

The process of checking the backpack for the availability of each object on the list is **test case execution**.

An essential part of the test case is the expected result—for instance, "Large bottle of Sapporo". Thus, a test case can consist of only an expected result.

Please, note that today we'll talk about **formal side** of test case; i.e., test case structure, useful attributes; formatting, etc. In future lectures, we'll learn about **actual side** of test case; i.e., about test case content that would have high probability to find a bug.

Test Case Structure

The problem with a test case having **only the expected result** is that this piece of information (expected result) alone might not be sufficient to perform the test case execution. In the case of a large bottle of Sapporo, we can just unzip the backpack and look inside. In the case of software, as a rule, we need **an instruction** on how to reach an actual result.

Example

Our red-hot start-up, www.sharelane.com, has a new employee; let's call him Samuel Pickwick. Some friendly fellow hands him the following one-line test case:

"Payment can be made by Visa."

Mr. Pickwick's brain immediately generates two perfectly reasonable questions:

- "How can I do testing if I don't have a Visa card and don't know where to get one?"
- "How can I be sure that a payment was really made even if I had that Visa card?"

One thing that is more or less clear is the purchase process (create new account and log in, find a book to buy, put it into the shopping cart, do checkout), but even with this common knowledge, the test case execution will be stuck at the point where the tester enters credit card information.

In other words, that test case cannot be executed by Mr. Pickwick without someone helping him. This situation sucks because:

- It adds even more stress to Mr. Pickwick's first days at his new company.
- Mr. Pickwick MUST interrupt the work of other tester (-s) to get the information he needs.

Here is another example.

Example

Our red-hot start-up, www.sharelane.com, has a new employee; let's call him Samuel Pickwick. Some friendly fellow hands him the following test case:

SL

1. Go to <http://main.sharelane.com>.
2. Click link "Test Portal".
3. Click link "Account Creator".
4. Press button "Create new user account".
5. Copy email to the clipboard.
6. Go to <http://main.sharelane.com>.
7. Paste user email into textbox "Email".
8. Enter "1111" into textbox "Password".
9. Press button "Login".
10. Enter "expectations" into textbox "Search".
11. Press button "Search".
12. Press button "Add to Cart".
13. Click link "Test Portal".
14. Click link "Credit Card Generator".
15. Select "Visa" from drop-down menu.
16. Press button "Generate Credit Card".
17. Copy card number to the clipboard.
18. Go to <http://main.sharelane.com>.
19. Click link "Shopping cart".
20. Press button "Proceed to Checkout".
21. Select "Visa" from drop down menu "Card Type".

22. Paste card number into text box "Card Number".
23. Press button "Make Payment".
24. Write down order id: ____.
25. Click link "Test Portal".
26. Click link "DB Connect Utility".
27. Make database query:
select result from cc_transactions where id = <order id>;

Expected result: "10"

In our last example (we'll refer to it as "Test Case with Credit Card"), the expected result (**ER**) was preceded by steps that were supposed to give the test case executor an actual result (**AR**). These steps can be also called the "**procedure**".

Here is an analogy for you:

- The procedure is a set of steps on a staircase.
- The expected result is a certain object we are supposed to find if we climb to the top of that staircase.
- The actual result is what we actually find when we reach the top of the staircase.

Brain positioning

If we apply the computer science concept of input/output, we'll have this:

- The procedure is the *instruction about input*.
- The execution of the procedure is the *act of input*.
- The expected result is the *expected output*.
- The actual result is the *actual output*.

Results Of The Test Case Execution

The test case execution is finished once we have compared the actual and expected results. This comparison can give us one of two results:

1. PASS—i.e., AR equals ER
2. FAIL—i.e., AR does not equal ER (**bug?** - see below)

Brain positioning

If test case execution result is FAIL it doesn't necessarily mean that we found a bug.

Here is typical start-up situation: product manager communicated certain change with programmer, but none of them communicated that change with test engineer.

So, code is going to be **legitimately** changed, but test engineer will not be aware of it and test case execution will end up with FAIL. Thus the FAIL would be caused not by bug, but by the fact that **expected result known to tester** is wrong.

Sometimes we have a situation where the test case execution is blocked, so we cannot get to the actual result. For example, in Test Case with Credit Card, we can be stuck at Step 23 if the *Make Payment* button doesn't exist on the Web page. In that case, we would file a bug about the absence of the *Make Payment* button, and we would delay the execution of this test case until that bug is fixed.

Useful Attributes Of The Test Case

In addition to essential parts like expected result and procedure, test case can have a number of useful attributes that make our lives easier:

- Unique ID
- Priority
- IDEA
- SETUP and ADDITIONAL INFO
- Revision History

UNIQUE ID

This is an extremely important attribute. A test case without an ID is like a house without an address. The test case ID should be unique, not only within the concrete test suite which contains that test case, but also within all test suites inside the company. The rationale for this is that, as time goes by, it will be necessary to keep test case statistics; update, remove, or move test cases between test suites, etc.

PRIORITY

The test case priority reflects how important this test case is. The priority is graded from 1 to n, with 1 being the highest priority. I think that it's rational for 4 to be the lowest priority.

For example, the test case that checks if the Make Payment button works must be P1, and the test case that checks the color of the Web link "Contact" is P4.

Why do we need to prioritize test cases? Let's assume that we have two test cases, P1 and P3. We have time enough to execute only one of them. Which one do we pick? P1, of course!

Test case prioritization is especially important during regression testing, i.e. testing of old functionalities (we'll talk about regression testing a lot in this Course).

Question: How do we assign a test case priority?

Answer: As a rule, the author of the test case decides the importance of the thing being checked by that test case.

IDEA

This is a description of **what** we check for a particular test case.

Example

In the Test Case with Credit Card, ER is "10." What info do we get by looking at "10"? Nothing except two digits. However, the developers of www.sharelane.com have developed the following process to create a code for the result of a transaction:

- SL** - The first digit corresponds to the internal id of the credit card: "1" for Visa, "2" for MasterCard, and "3" for AmEx (see table cc_types: Test Portal>DB>Data>cc_types)
- The second digit is the success code: "0" for a successful transaction, and "1" for a failed transaction.

Now we have the full understanding what "10" means: the transaction with Visa was successful.

The problem is that **even the author** of the Test Case with Credit Card can forget what "10" means, because in the end, everything is perishable and forgettable (except, of course, high school sweetheart Anna V.) That's why at the beginning of a test case its author should put in human language the following phrase: "Payment can be made by Visa," so everyone who must execute that test case will immediately understand what is being tested.

SETUP AND ADDITIONAL INFO

A culinary recipe consists of two parts:

1. A list of ingredients
2. Instructions on how to fry, steam, or bake lucky guys from number 1.

The first part of a recipe is needed to enable a chef to

- see beforehand AND
- in one place

all the necessary components of the recipe and have them ready once the time comes to prepare it. This is a very practical first step.

The setup part of a test case can include:

- **Information about existing (also called "legacy") user accounts** or instruction on how to create a new user account.
- **Data used in the test case**; e.g. default password for testing.
- **SQL queries** (also called "*SQL statements*"); i.e., commands used to interact with the database (DB). *SQL* (*pronounced "sequel"*) stands for *Structured Query Language*.
- **Comments to help the tester**; e.g., instructions about where to get the software needed for the test case execution (e.g. Windows software to connect to the DB).
- **Other items that can ease the execution and maintainability of the test case** (we'll talk about maintainability in a minute).

REVISION HISTORY

In order to have data about the birth and history of the modifications for each test case, we create a mini-journal of any changes where we record: When, Who, Why and What.

Details:

- ✓ **Created (date/name)** – when the first version of this test case was created and who created it.
- ✓ **Modified (date/name)** – when any change to the test case took place and who was responsible for the change
- ✓ **Reason** – why the test case was changed and what was changed

You can view/download a test case template (as well as all the other documents mentioned in this Course) from Downloads on qatutor.com.

Let's create the Test Case with Credit Card using all mentioned test case attributes:

IDEA: Payment can be made by Visa	TC ID	CCPG0001
	Priority	1
SETUP and ADDITIONAL INFO		
Password: "1111" (this is default unchangeable password for all user accounts) Search keyword: "expectations"		
!!!Go to Test Portal>Helpers>DB Connect Utility to run SQL query below.		
SQL1: select result from cc_transactions where order_id = <order id>;		
Created (date/name): 11/17/2004/O.Ferguson	Reason: new way to verify that credit card transaction was successful.	
Modified (date/name):	Reason:	
SL	✓ 10 1. Go to http://main.sharelane.com. 2. Click link "Test Portal". 3. Click link "Account Creator". 4. Press button "Create new user account". 5. Copy email to the clipboard. 6. Go to http://main.sharelane.com. 7. Paste user email into textbox "Email". 8. Enter password into textbox "Password". 9. Press button "Login". 10. Enter search keyword into textbox	

<p>"Search".</p> <ol style="list-style-type: none"> 11. Press button "Search". 12. Press button "Add to Cart". 13. Click link "Test Portal". 14. Click link "Credit Card Generator". 15. Select needed card from drop-down menu. 16. Press button "Generate Credit Card". 17. Copy card number to the clipboard. 18. Go to http://main.sharelane.com. 19. Click link "Shopping cart". 20. Press button "Proceed to Checkout". 21. Select appropriate value from drop down menu "Card Type". 22. Paste card number into text box "Card Number". 23. Press button "Make Payment". 24. Write down order id: ____. 25. Run SQL1 	
---	--

Data-Driven Test Cases

The main advantage of our brand-new, shiny Test Case with Credit Card is that we don't have to modify the steps necessary to test other credit cards if we want to test them in a similar way. One thing we do need to modify, however, is the test case DATA. So, if we want to test two more cards the same way we just tested Visa, we

- Do “copy” one time
- Do “paste” two times
- In both new test cases, we don't change any steps. Instead, we change only the two (*italicized* and **bolded**) values living in the header and the expected result (remember that we also have to change the test case ID and any other attributes):

Visa
10

This type of test case is called **data-driven**, because the **data and the steps using that data are not mixed up in one pile, but separated and linked with each other**.

Maintainability Of Test Cases

The new Test Case with Credit Card looks good: it's nicely formatted, it's data-driven, and it includes useful test attributes. The problem is that the Web site, and especially the part called “User Interface” (UI), is subject to frequent changes.

Example

The “Login” button from Step 9 can easily be renamed “Sign-in.” Hence, if we have three test cases, we have to make three changes. But what if we have 500 test cases where the “Login” button is mentioned, and what if these test cases are scattered among many test suites - like my former classmates are scattered throughout the world? You might say, “Not a big deal. It’s easy to guess.” But as time goes by, we could have hundreds of small changes like that! Gradually, our test cases will either start to lose their practical value (because of the difficulty in executing them), or they will start to consume too much time for maintenance.

So, the name of the issue is “maintainability”; i.e., the simplicity and ease of changing a test case to reflect changes in the software. **If we don’t think about test case maintainability, we aren’t thinking about tomorrow.**

Now, let’s split the steps of the Test Case with Credit Card into logical modules:

LOGICAL MODULE	TEST CASE STEPS
1. Create new user.	1. Go to http://main.sharelane.com . 2. Click link "Test Portal". 3. Click link "Account Creator". 4. Press button "Create new user account". 5. Copy email to the clipboard.
2. Login.	6. Go to http://main.sharelane.com . 7. Paste user email into textbox "Email". 8. Enter password into textbox "Password". 9. Press button "Login".
3. Find a product.	10. Enter search keyword into textbox "Search". 11. Press button "Search".
4. Add product to Shopping Cart.	12. Press button "Add to Cart".
5. Generate Credit Card	13. Click link "Test Portal". 14. Click link "Credit Card Generator". 15. Select needed card from drop-down menu. 16. Press button "Generate Credit Card". 17. Copy card number to the clipboard.
6. Do Checkout.	18. Go to http://main.sharelane.com . 19. Click link "Shopping cart". 20. Press button "Proceed to Checkout". 21. Select appropriate value from drop down menu "Card Type". 22. Paste card number into text box "Card Number". 23. Press button "Make Payment".
7. Get Order ID.	24. Write down order id: _____.
8. Query DB.	25. Run SQL1

Let's see how we can make this test case more maintainable:

1. Create new user

We do have to tell tester how to create new user account using test tool Account Creator (Test Portal>Helpers>Account Creator), but why shall we put the same steps over and over again every time when new account needs to be created? Instead, what if we:

- select blocks of the steps which will be repeated in many test cases,
- put those blocks into an external document and
- put a reference to that external document in those test cases where our steps used to be.

For example:

Create New Account

Using this approach, we can save ourselves enormous effort, because if test case steps have to be changed, we need to make that change in ONE place only! How cool is that!

BTW

SL Create New Account is a link to the corresponding Web page located on Intranet* site Test Portal. See Test Portal>More Stuff>QA KnowledgeBase>Create New Account.

*internal company network

2. Login

It's easy to guess where to enter email/password and which button to press to login. So, we can get rid of unnecessary steps (Steps 6, 7, 8, 9).

Note that we aren't testing the login process here – we have separate test cases for that. Now we are simply using login as one of the steps to execute our Test Case with Credit Card.

3. Find a product

An approach from section "2. Login" above is applicable here too.

What else can we do here? Let's say that someone accidentally deleted book "Great Expectations" from the DB. What shall we do now? Blame the world for being unfair and whine that we are blocked? No.

SL We simply prevent that situation by looking into DB table "books" to see available book titles (Test Portal>DB>Data>books). So, during test case execution we'll just search for a keyword from **any** title available in table "books". Let's create a QA KnowledgeBase article for that: Find Search Keyword (Test Portal>More Stuff>QA KnowledgeBase>Find Search Keyword)

4. Add product to Shopping Cart

An approach from section "2. Login" above is applicable here too.

5. Generate Credit Card

SL Let's create QA KnowledgeBase article [Generate Credit Card](#) (Test Portal>More Stuff>QA KnowledgeBase>Generate Credit Card).

6. Do Checkout

SL Let's create QA KnowledgeBase article [Do Checkout](#) (Test Portal>More Stuff>QA KnowledgeBase> Do Checkout).

7. Get Order ID and 8. Query DB.

Let them be.

Let's recap the measures we took to improve the maintainability of test cases:

1. **Make the test case data driven.**
2. **Don't include steps for obvious, easy-to-guess scenarios.**
3. **Don't give concrete details if they don't matter during the execution of the test case—e.g., the search keyword.**
4. **Take repeated scenarios, move them into QA Knowledge Base, and put a link to that QA KB page into the test case.**

IDEA: Payment can be made by <i>Visa</i>	TC ID	CCPG0001
	Priority	1
SETUP and ADDITIONAL INFO		
Go to Test Portal>More Stuff>QA KnowledgeBase to "See QA KB"		
<i>In order to run SQL1 go to Test Portal>Helpers>DB Connect Utility.</i>		
SQL1: select result from cc_transactions where order_id = <order id>;		
Created (date/name): 11/17/2004/O.Ferguson	Reason: new way to verify that credit card transaction was successful.	
Modified (date/name): 02/03/2005/I.Newsome	Reason: modified steps to improve test case maintainability	
SL	<i>✓ 10</i>	
1. <u>Create New Account (See QA KB)</u> 2. Login 3. <u>Find Search Keyword (See QA KB)</u> 4. Do search 5. Add found book to the Shopping cart. 6. <u>Generate Credit Card (See QA KB)</u> 7. <u>Do Checkout (See QA KB)</u>		

- | | |
|-------------------------------|--|
| 8. Write down order id: _____ | |
| 9. Make DB query: SQL1 | |

BTW

For execution:

- Test cases can be printed out on paper.
- Test cases can be viewed on a monitor as opened MS Word or MS Excel files (or whatever files we use for test cases).
- Test cases can be viewed on a monitor as a part of special test case management software.

Remember, Create New Account, Find Search Keyword, Generate Credit Card and Do Checkout are an HTML links. If you have printed out the test case, you cannot click the link (and you might have problems understanding that it is a link, especially if the printout is in black and white). To solve this problem, you can format those links in a special way. For example,

- use special font formatting (**bold**/*italic*/underline/font type/font size) and/or
- use magic words that will follow EVERY reference to QA Knowledge Base, like "See QA KB".

So, your link would look like: *Do Checkout (see QA KB)*.

The Number Of Expected Results Inside One Test Case

Our test case verifies only one concrete thing: "**Payment can be made by Visa**" and in an ideal situation there is only one ER. If I was a test theoretician, I'd tell you to never include more than one ER in a test case, but I'm no theoretician and I promised you to teach you real stuff. Here is a live example for you:

Example

Let's assume that according to item 12.b. of the document "Code Design for the Spec #6522", we can say that a payment with Visa is successful if—and only if—two conditions are met:

1. In the DB, the column result of the table cc_transactions has a value of "10" for the record with our Visa transaction.
2. The credit card balance is reduced by the amount equal to the amount of the payment.

So, in order to verify only **one concrete thing**, we have to check to see if the reality meets **two expected results**.

We have two ways:

1. Split the test case IDEA into two IDEAs and create two test cases OR
2. Don't change the test case IDEA and have two ERs in one test case. The test case would pass if, and only if, BOTH ARs match the corresponding ERs. In all other cases, the test case would fail.

Here is an example of the second way:

IDEA: Payment can be made by <i>Visa</i>	TC ID	CCPG0001
	Priority	1
SETUP and ADDITIONAL INFO		
Go to Test Portal>More Stuff>QA KnowledgeBase to "See QA KB"		
<i>In order to run SQL1 go to Test Portal>Helpers>DB Connect Utility.</i>		
SQL1: select result from cc_transactions where order_id = <order id>;		
How to get credit card balance: Test Portal>Helpers>Credit Card Balance Viewer		
Created (date/name): 11/17/2004/O.Ferguson	Reason: new way to verify that credit card transaction was successful.	
Modified (date/name): 02/03/2005/I.Newsome	Reason: modified steps to improve test case maintainability	
Modified (date/name): 02/15/2005/I.Newsome	Reason: modified steps and added second expected result to verify that card balance was reduced.	
SL 1. <u>Generate Credit Card (See QA KB)</u> and copy card number into the temporary text file. 2. Get credit card balance and write it down:  3. <u>Create New Account (See QA KB)</u> 4. Login 5. <u>Find Search Keyword (See QA KB)</u> 6. Do search 7. Add found book to the Shopping cart. 8. Copy card number from the temporary text file and <u>Do Checkout (See QA KB) !!!</u> While doing it, write down amount of payment: 9. Write down order id: _____ 10. Make DB query: SQL1	✓ 10	
11. Get credit card balance and write it down: 	✓ Step 2 - Step 8	

How will we execute this test case? Simple:

- Execute first 10. Compare the AR and ER.
- Execute Step 11. Compare the AR and ER.

The test case would pass if, and only if, both conditions are true:

1. The AR after Step 10 equals "10" **AND**
2. The AR after Step 11 equals "Step 2 - Step 8"—i.e., the result of subtracting the amount of payment (Step 8) from the card balance before payment (Step 2).

In theory, it would be cleaner to split the test case into two parts and create two test cases:

1. IDEA: "The correct success code is inserted into the DB if Visa is used"
2. IDEA: "The correct amount is debited from the Visa balance during Checkout"

If possible, it's better to create two test cases, BUT in the vast majority of situations, it's more practical to combine two or more ERs into *one* test case. The rationale:

- You probably will not have time to write, execute, and maintain TWO test cases. *If we have only one situation where two ERs can be combined, writing, executing, and maintaining two test cases are not a problem. But what if you have HUNDREDS of situations like that, and thus hundreds of extra test cases?*

- The saved time and effort can be spent writing, executing, and maintaining at least one more test case. *As time goes by, you'll create HUNDREDS of additional test cases that would make a huge difference on how thoroughly your application is tested.*

I've been working with test cases that include more than one ER for many years. And the software that I've been testing is very complex financial software of PayPal that processes billions of dollars worth of payments. So, with all confidence I can tell you that two or more ERs in one test case is normal practice, and your effectiveness will only increase if you follow this approach for your projects.

Very often, when you have two or more ERs, you have to check for:

- a certain value on the Web page and/or
- a certain value in the DB and/or
- a certain value in the log file

BTW

A log file is simply a text file in a specified format that keeps track of particular events.

For example, a programmer can write a code that appends a text file in the following format every time someone successfully pays with credit card:

<UNIX timestamp*>,<card type: 'V' for Visa, 'M' for MasterCard, 'A' for AmEx>,<4 last digits of card number>,<amount in cents>,<order id>

Here is an example of the content of such log file:

1184970195,V,8277,1000,762
1184970197,M,7844,1000,763

SL See Test Portal>Logs>cc_transaction_log.txt

*A UNIX timestamp is the number of seconds that have passed since midnight on January 1st, 1970. Programmers often use a UNIX timestamp to log the **precise time of a specific event**, like the creation of new user account.

This means that you need to check both the **front end** and the **back end** of your software to determine if that test case passed or not.

Brain positioning

The **front end** is the interface that customers can see and use; e.g., text, images, buttons, links. Compared to a car, the front-end pieces are items like the steering wheel and the dashboard.

The **back end** is the software and data that are hidden from the user; e.g., the Web server, the DB, the log files, etc. Compared to a car, the back-end pieces are items like the engine and the electrical circuit of the car.

One last note about two or more expected results: We looked at a situation requiring a step ("11. Get credit card balance") between ER1 and ER2. In practice, you'll also encounter situations when you'll have two or more ERs in one place; in other words, your ERs will not be separated by steps. Here is a schematic steps-ER portion of such a test case:

1. Do something	✓ Verify something
2. Do something	✓ Verify something
3. Do something	

Bad Test Case Practices

Let's take a look at 3 bad test case practices and learn how to avoid them.

1. Dependency between test cases
2. Poor description of the steps
3. Poor description of the IDEA and/or expected result

1. DEPENDENCY BETWEEN TEST CASES

"Dependency" is the opposite of "independency." An independent test case is a test case that does not rely on any other test cases.

Example

Test Case #1

The steps:

1. Enter living room.
2. Head for large leather chair.
3. Open **right** external pocket of backpack.

Expected result: huge beer mug

Test Case #2

The steps:

1. Enter living room.
2. Head for large leather chair.
3. Open **left** external pocket of backpack.

Expected result: potato chips

As we can see, Steps 1 and 2 are the same in both test cases. Hence, there can be a temptation to improve the steps (especially after learning about test case maintainability). Let's do this and see what happens.

Example**Test Case #1**

The steps:

1. Enter living room.
2. Head for large leather chair.
3. Open **right** external pocket of backpack.

Expected result: huge beer mug.

Test Case #2

The steps:

1. See Steps 1 and 2 from Test Case #1
2. Open **left** external pocket of backpack

Expected result: potato chips

Does Test Case #2 look more modular now? Maybe.

Should we use this practice? **NEVER.**

Why not?

What if:

- Test Case #1 is deleted because we don't need it anymore (e.g., no more beer during fishing)?
- Steps 1 and 2 of Test Case #1 are changed (e.g., huge beer mug is put into another backpack, which is in the kitchen)?

In both cases, it's not clear how to execute Test Case #2 because:

- We either don't know the steps at all, or
- The steps that we have are not correct.

Another frequent situation occurs when there is an **assumption** that the software and/or DB is brought to a certain state because previous test cases have been executed.

Example

Test Case #1

The steps:

1. Set the count of MasterCard transactions in the DB to 0.
2. Buy a book using MasterCard.

Expected result: the transaction success code is “20”.

Test Case #2

The steps:

1. Buy a book using MasterCard.

Expected result: the count for MasterCard transactions is “2”.

So, Test Case #2 was written with the assumption that before its execution:

- Test Case #1 had already been executed.
- The code tested in Test Case #1 doesn't have any bugs, so all data is generated in the correct way.

What is wrong with this approach? In real life, many things concerning Test Case #1 might happen. For example:

- Test Case #1 might be deleted.

- Test Case #1 might be modified so that it creates transactions with a different credit card; e.g., Visa.
- Test Case #1 might not create any transactions simply because the code is broken.

In all three cases, the execution of Test Case #2 will at least lead to confusion because its execution relies on data generated during the execution of Test Case #1.

Thus, a good test case:

- Doesn't refer to other test cases.**
- Doesn't depend on "tracks" left by the execution of other test cases.**

Hence, if we have ten test cases which are independent, we can execute them in any order: Test Case #10, Test Case #2, Test Case #7, etc.

Sometimes the repetition of steps in several test cases seems redundant, but the advantages of creating independency in the test cases outweigh the inconvenience of the copy-paste operation.

2. POOR DESCRIPTION OF THE STEPS

Imagine that after visiting beautiful San Francisco, you have to drive to Los Angeles for your high-school reunion. You ask your friend Arnold who lives in San Francisco to write down driving directions from Haight Street in San Francisco to Vine Street in Los Angeles.

a. What if Arnold gives you these directions:

1.	Take US-101*
2.	Take I-80
3.	Take I-580
4.	Take I-5
5.	Take CA-170
6.	Take US-101
7.	Take exit 9A

*number of the freeway (high-speed road).

I sincerely doubt that anyone who is unfamiliar with driving from SF to LA can use this. The problems will start when you reach US-101 (Step 1), because you will not know whether to go North (**N**) or South (**S**) to end up at I-80. Here is a situation **where you have steps, but these steps lack clarity, and thus it's hard to use them.**

b. What if you are given driving directions like these:

1.	Take US-101 S
2.	Take I-80 E

3.	Take I-580 E
4.	Take I-5 S
5.	Take CA-170 S
6.	Take US-101 S
7.	Take Exit 9A

This set of directions is much better, because once you approach the freeway it's clear what direction (North, South, East, or West) to take.

But there is another problem. How would you first get to US-101 S if you don't know San Francisco? Yes, you could ask some unconcerned hippie on Haight Street, or you could follow the signs on the electricity poles, but what is the guarantee that you'll merge onto US-101 at the part that is north of I-80, so that Step 1, "Take US-101 S," would eventually bring you to Step 2? In the driving directions above, it's not clear how to reach Step 1, because Arnold:

- had an **assumption** that it's not a big deal to get to US-101, and
- **he knows** that the closest part of US-101 is to the north of I-80.

I suggest that you fire Arnold as your how-to-get-there consultant and use Google maps instead:

1.	Head north on Divisadero St toward Page St
2.	Turn right at Oak St
3.	Turn right at Octavia Blvd
4.	Slight right at Central Fwy/US-101 S (signs for US-101 S)
5.	Take the exit on the left onto I-80 E toward Bay Bridge/Oakland
6.	Take the exit onto I-580 E toward CA-24/Downtown Oakland/Hayward-Stockton
7.	Merge onto I-5 S
8.	Slight right at CA-170 S (signs for Hollywood/CA-170 S)
9.	Merge onto US-101 S
10.	Take Exit 9A for Vine St
11.	Turn right at Vine St

Very nice! The lesson here is that all the steps should be clear and concrete.

You should always keep in mind that:

- Whatever is obvious for you now can become absolutely unclear in several months.**

For example, unexplained abbreviations (e.g., "DCBT"), compacted wording (like "svrl" instead of "several") and other similar items can become Egyptian hieroglyphs even to the author of the test case, so it will be easier to create and execute a new test case rather than cut through the jungle of these vague steps.

BTW, "DCBT" stands for ... "do cross-browser testing." Did you guess it? No? Neither did I, so I had to call the test case author (who had already left the company) to find out what he meant by "DCBT."

- A test case that can be executed only by its author should be publicly denounced, burnt, and its ashes thrown into the ocean to make sure that no one will ever try to execute that test case again.

Why? Very simple. What if the test case author gets sick, takes a vacation, departs from the company, or even carelessly dies, leaving behind an unexecutable test case? **Each test case should be created with the thought that another person will have to execute it in the future.**

But even with all of this, try to remember that excessive details are bad for the maintainability of test cases. So, my friends, let's try to find a middle way.

3. POOR DESCRIPTION OF THE IDEA AND/OR EXPECTED RESULT

Both principles that we just talked about:

- The future can erase from one's memory those things which are crystal clear today
- You don't create test cases for yourself; you create them for the company

are also applicable for describing an IDEA and/or ER. Here are the nuances:

a. I don't recommend making a reference to the external document in the descriptions of an IDEA and/or ER.

When we've talked about sending repetitive scenarios to the QA Knowledge Base, the reason for doing so was to improve the maintainability of the test cases. In the case of IDEA and ER, it's a different story.

Example

Would it be **comfortable** to execute a test case with this IDEA:

"In this test case, we check item 21.b of the spec #3934 "Adding credit card to user's account.""

Or how about a test case with this ER?

"Verify that the value of the last step equals the result of calculation according to the first formula from the top on page 233 of the book Stock Market Finances where X is the value from Step 2, Y is the value from Step 24, and Z is a value in the column "Number of Calls" corresponding to the value from Step 11 in Table 1 of the Spec #S122709."

I doubt that you would be happy to execute a test case like that. I got sweaty even reading that ER!

b. We have to remember that the purpose of an IDEA is TO EXPLAIN.

If the section IDEA is empty or modestly states "10," everyone who executes this test case will have to spend several minutes of his or her time to clarify what the heck he or she is testing here.

c. We have to remember that the ER is the information we use (along with the AR) to make a decision as to whether a test case passed or failed.

Therefore, precision and clarity play a most critical role in ER.

Example

The Expected Result states: "Verify if error message is displayed." Mr. Wei executes a test case and sees that the error message is displayed. But what if the error message says, "Please provide your Social Security Number," while according to the spec it should say, "Your Social Security Number is invalid"? In this case, the bug will be missed. It wouldn't be Mr. Wei's fault, but the fault of the tester who created that ER.

BTW

Please note that I don't advocate that the test case author has to necessarily provide the concrete text of an error message. For example, in a startup environment, the text of an error message can be changed 3 times a week, and, so for maintainability purposes, it makes much more sense to create this kind of ER: "Verify that the error message about an invalid SSN is displayed," rather than to put the concrete text of the error message and then spend time for maintenance every time the wording of that error message is changed.

Another example of a bad ER: "*Everything works*".

Test Suites

As we already know, a test case helps to check only one concrete thing (which is explained in the IDEA section of the test case). Each spec might be a source for a huge number of ideas for testing, so to test the code that is written according to the spec, we need many test cases.

A combination of test cases that check

- the concrete part of our project (e.g., "Shopping Cart") and/or
- the concrete spec (e.g., Spec #1455 "Privacy Rules")

is called a **test suite**. As a rule, all test cases that belong to the same test suite are placed in the same document; for instance, an MS Word file.

Here is a real life scenario:

1. A product manager gives me a new spec.
2. I create a new MS Word file into which I copy and paste the test case templates (check the Downloads section of qatutor.com).

3. I fill up those templates with content.
4. I execute those test cases, making necessary modifications if necessary.
5. If it makes sense, I integrate this suite into the main test suite where I store other test cases that test other aspects of the same functional area: e.g., "Purchase with Credit Cards."

Let's look at an example:

Example

Our super-duper Web site, www.sharelane.com, accepts Visa and MasterCard. We have a test suite called "Checkout with Credit Cards" that we execute before each release to check if new bugs were introduced to old functionalities (i.e., checkout with Visa and MasterCard). BTW, this type of testing is called *regression testing*.

This test suite was based on spec #1211 and has test cases to test Checkout with Visa and MasterCard.

For the new release, the product manager created spec #1422. This spec introduces another credit card that we are going to accept: American Express (AmEx).

First we create a new test suite, "Checkout with AmEx," and fill it up with new test cases. Then we execute those test cases and modify them if needed.

After that, because the test cases from "Checkout with AmEx"

- are cleaned up and
- check the same functional area as the main test suite (i.e., "Checkout with Credit Cards")

it is logical for us to integrate the test suite "Checkout with AmEx" into the test suite "Checkout with Credit Cards."

Brain positioning

Nobody expects your test cases to be perfect right after you've written them. Here are some of the reasons why:

1. A spec can have bugs.
2. A spec can have some omissions and other spec maladies.
3. A spec can be changed without anyone notifying you.

Also, if you are writing test cases for a code that is not available (e.g., you generate your test cases while the programmer is writing the code) your knowledge of functionalities is THEORETICAL. So, even if a spec is perfect, there is always a chance that you have made a mistake or an omission when writing your test cases **just because you didn't have a chance to have PRACTICAL interaction with the code**. This part of a tester's job—creating test cases for code that doesn't exist—can be really confusing for beginners, but don't worry, **your experience will heal your confusion**.

So, there are many reasons why freshly baked test cases are not perfect. The best situation is when a new test case is first executed by its author: person who totally understands whatever was typed into the test case. During first execution the author usually:

- adds more test cases to check more things;
- changes the test case content, such as the ER; e.g., if he or she didn't understand spec correctly while creating that test case;
- deletes duplicate test cases (test cases that check exactly the same thing);
- makes test cases easier to maintain;
- polishes test cases to make their content more precise and crystal clear.

Here is a header that we can insert at the beginning of our test suite:

Author	Spec ID	PM	Developer	Priority
OVERVIEW				
GLOBAL SETUP AND ADDITIONAL INFO				
REVISION HISTORY				

Author — the author (-s) of test cases.

Spec ID — the unique ID of the spec that describes tested functionalities; this ID should be an HTML link to the spec on our Intranet.

PM (Product Manager) — person (-s) who wrote the spec.

Developer — the programmer (-s) responsible for writing a code according to the spec.

Priority — the priority of the test suite (usually from 1 to 4, with 1 being most important); usually the priority of the test suite matches the priority of the spec.

In the **OVERVIEW**, we usually give the general idea about what we're testing with this test suite: e.g., "Here we test the various scenarios when a customer uses Visa and MasterCard during Checkout." The **OVERVIEW** represents the IDEA of the test suite.

The purpose of the **GLOBAL SETUP AND ADDITIONAL INFO** section is to educate the person executing the test suite about technicalities that will be helpful during the execution of the test cases included into that test suite. It's just like the corresponding section of test cases, but in the test case, we put helpful info about that **particular** test case, while in a test suite we use this section to put information that can be applied towards the execution of **two or more test cases**.

Here is the content of the file, credit_card_payments.doc, which includes the test suite "Checkout with Credit Cards":

Checkout with Credit Cards (TS7122)*

*(TS7122) — each test suite must have a unique ID.

Author	Spec ID	PM	Developer	Priority
O. Ferguson, I. Newsome	<u>2011</u>	R. Gupta	G. Snow	1
OVERVIEW				
This test suite checks whether users can pay with Visa and MasterCard.				
GLOBAL SETUP AND ADDITIONAL INFO				
Go to Test Portal>More Stuff>QA KnowledgeBase to "See QA KB"				
<i>In order to run SQL1 go to Test Portal>Helpers>DB Connect Utility.</i>				
SQL1: select result from cc_transactions where order_id = <order id>;				
How to get credit card balance: Test Portal>Helpers>Credit Card Balance Viewer				
REVISION HISTORY				
11/17/2004 - Added CCPG0001 and CCPG0002 - O.Ferguson 02/03/2005 - Revised CCPG0001 and CCPG0002 - I.Newsome 02/15/2005 - Revised CCPG0001 and CCPG0002 - I.Newsome				

IDEA: Payment can be made by <i>Visa</i>	TC ID	CCPG0001
	Priority	1
SETUP and ADDITIONAL INFO		
Created (date/name): 11/17/2004/O.Ferguson	Reason:	new way to verify that credit card transaction was successful.
Modified (date/name): 02/03/2005/I.Newsome	Reason:	modified steps to improve test case maintainability
Modified (date/name): 02/15/2005/I.Newsome	Reason:	modified steps and added second expected result to verify that card balance was reduced.
1. <u>Generate Credit Card (See QA KB)</u> and	✓	10

<p>copy card number into the temporary text file.</p> <p>2. Get credit card balance and write it down:</p> <p><u>3. Create New Account (See QA KB)</u></p> <p>4. Login</p> <p><u>5. Find Search Keyword (See QA KB)</u></p> <p>6. Do search</p> <p>7. Add found book to the Shopping cart.</p> <p>8. Copy card number from the temporary text file and <u>Do Checkout (See QA KB) !!!</u> While doing it, write down amount of payment:</p> <p><u>9. Write down order id: _____</u></p> <p>10. Make DB query: SQL1</p>	
11. Get credit card balance and write it down:	✓ Step 2 - Step 8

IDEA: Payment can be made by MasterCard	TC ID	CCPG0002
	Priority	1
SETUP and ADDITIONAL INFO		
Created (date/name): 11/17/2004/O.Ferguson	Reason: new way to verify that credit card transaction was successful.	
Modified (date/name): 02/03/2005/I.Newsom	Reason: modified steps to improve test case maintainability	
Modified (date/name): 02/15/2005/I.Newsom	Reason: modified steps and added second expected result to verify that card balance was reduced.	
<p>1. <u>Generate Credit Card (See QA KB)</u> and copy card number into the temporary text file.</p> <p>2. Get credit card balance and write it down:</p> <p><u>3. Create New Account (See QA KB)</u></p> <p>4. Login</p> <p><u>5. Find Search Keyword (See QA KB)</u></p> <p>6. Do search</p> <p>7. Add found book to the Shopping cart.</p> <p>8. Copy card number from the temporary text file and <u>Do Checkout (See QA KB) !!!</u> While doing it, write down amount of payment:</p> <p><u>9. Write down order id: _____</u></p> <p>10. Make DB query: SQL1</p>	✓ 20	
11. Get credit card balance and write it down:	✓ Step 2 - Step 8	

Please note the following:

1. Setup/additional info that can be applied to more than one test case (e.g., SQL1) has been moved to the GLOBAL SETUP and ADDITIONAL INFO section of the test suite.
2. Data that is different between test cases CCPG0001 and CCPG0002 (e.g., expected result) has been typed in a bold and italic font. This was done to attract attention to the differences between similar looking test cases.

It's a great idea to use the formatting **FUNctionalities** of your text editor

- to stress things that deserve special attention and
- to make it easier for reader to follow you.

Let's move along.

Our manager gives us spec #1422 "Checkout with AmEx" written by product manager Y.Wang. We create a new file, amex_payments.doc, and after we've created and improved our test cases (or only 1 test case like in this test suite), we have:

Checkout with AmEx (TS7131)

Author	Spec ID	PM	Developer	Priority
I. Newsome	<u>1422</u>	Y. Wang	G. Snow	1
OVERVIEW				
This test suite checks whether users can pay with AmEx card.				
GLOBAL SETUP AND ADDITIONAL INFO				
Go to Test Portal>More Stuff>QA KnowledgeBase to "See QA KB"				
<i>In order to run SQL1 go to Test Portal>Helpers>DB Connect Utility.</i>				
SQL1: select result from cc_transactions where order_id = <order id>;				
How to get credit card balance: Test Portal>Helpers>Credit Card Balance Viewer				
REVISION HISTORY				
05/15/2005 - Created AEPL0001 - I.Newsome				

IDEA: Payment can be made by <i>AmEx</i>	TC ID	AEPL0001
	Priority	1

SETUP and ADDITIONAL INFO	
Created (date/name): 05/15/2005/I.Newsone	Reason: new way to verify that credit card transaction was successful.
1. <u>Generate Credit Card (See QA KB)</u> and copy card number into the temporary text file. 2. Get credit card balance and write it down: <u>3. Create New Account (See QA KB)</u> 4. Login <u>5. Find Search Keyword (See QA KB)</u> 6. Do search 7. Add found book to the Shopping cart. 8. Copy card number from the temporary text file and <u>Do Checkout (See QA KB) !!!</u> While doing it, write down amount of payment: <u>9. Write down order id: _____</u> 10. Make DB query: SQL1	✓ 30
11. Get credit card balance and write it down:	✓ Step 2 - Step 8

Now, after we've executed Test Suite #**TS7131** (and, if needed, modified its test case), we can marry TS7122 and TS7131 and create an all new credit_card_payments.doc. Let's see what this looks like:

Checkout with Credit Cards (TS7122)

PART I: tests with Visa and MasterCard.

PART II: tests with AmEx.

PART I

<Insert header, CCPG0001 and CCPG0002 from old file credit_card_payments.doc>

PART II

<Insert header and AEPL0001 from old amex_payments.doc>

SL See TS7122 here: Test Portal>More Stuff>TS7122

Please note: we haven't changed:

- either the content (header and AEPL0001) from the file amex_payments.doc
- or the content (header, CCPG0001, and CCPG0002) of the file credit_card_payments.doc.

Theoretically, we could have created a combined header and/or changed the test case ID from "AEPL0001" to "CCPG0003" (to have continuous numbering in the same test suite), but we are not going to do that and below are the reasons why:

- The content of amex_payments.doc and the content of credit_card_payments.doc represent two independent modules in the sense of test case execution
- A unique ID is given to a test case once and for all; it's like a SSN (Social Security Number).

BTW

A unique ID for a test case can be

- generated automatically (you can do it yourself, or ask your developer to write a program which generates sequential numbers)
- generated manually; in order to do this, we have to create a simple convention (rule) inside the QA Department

Example

We can agree that an ID consists of two parts:

- The first part consists of letters, an abbreviation for the name of tested functionality
- The second part consists of numbers: e.g., from 0001 to 9999

The original test case ID is assigned by the author of the first test case in the test suite, and if a new test case (without an ID) is added to that test suite, the ID for this new test case is generated like this:

- The first part (letters) is taken from the old test cases
- The second part equals the maximum number of the second part among existing IDs plus 1

For example, the third test case to test Visa or MasterCard will have the ID CCPG0003.

BTW

CCPG stands for “Credit Cards Payments Global”

AEPL stands for “AmEx Payments Local”

Why have I chosen those names for the tested functionalities? Because those names sounded logical to me. Each company has its own standards in this regard. The main point here is to make sure that each test case has its own unique ID.

Example

Here is how to assign IDs to new test cases:

1. First, generate some test cases. No IDs are assigned.
2. Second, execute these new test cases. During the execution,

- delete those test cases not worthy of testing (e.g., duplicate test cases);
- add new test cases that would improve the testing.

3. Then, assign IDs to all survived test cases.

States Of A Test Case

State: "Created"

This is the very first edition of test case.

Example:

Created (date/name): 11/17/2004/O.Ferguson

State: "Modified"

Modifications are usually triggered

- by a change in the tested functionality (so, we need to modify a test case to reflect those changes);
- by an intention to improve a test case (e.g., to make the test case more maintainable).

Example:

Modified (date/name): 02/03/2005/I.Newsome

State: "Retired"

The test case becomes obsolete:

- if the functionality tested by it doesn't exist anymore (e.g., if we stop accepting MasterCard, the test case CCPG0002 is not needed anymore);
- in other cases when that test case doesn't make sense (e.g., when a test case is a duplicate).

I don't recommend simply deleting a retired test case. My rationale:

- There can be an "error in judgment," and a retired test case can turn out to be valid.
- A retired test case can be of use in the future.

A better strategy is to:

1. Create directory `retired_testcases` in the same directory where we store files with valid test cases

Directory for valid test cases - **C:/testcases**

Directory for invalid (retired) test cases - **C:/testcases/retired_testcases**

2. Create a file with the same name as the file with the test suite where we take the test cases from

File for valid testcases - **C:/testcases/credit_card_payments.doc**

File for invalid test cases - **C:/testcases/retired_testcases/credit_card_payments.doc**

3. Cut-and-Paste no-longer-needed test case to its new retirement home

Move CCPG0002

FROM: **C:/testcases/credit_card_payments.doc**

TO: ***C:/testcases/retired_testcases/credit_card_payments.doc***

In reality, we often invalidate the whole test suite, so we would just move the whole file with that test suite to the directory retired_testcases.

Sometimes there is a dilemma. Which is better:

- to modify an old test case OR
- to retire the old test case and create a new one?

Each situation is unique, but life teaches us that it's faster to build a new building rather than to completely restore an old one. So, if a test case requires a lot of changes—e.g., if I have to edit steps in many places—I usually take an empty test case template and start writing a new test case.

The Importance Of Creativity

All that we've talked about so far can be used in suggested form, but I want you to grasp the main thing:

Testing is a creative process and thus assumes a search for perfection.

So, you have to search until you find what's BEST for your concrete company and concrete situation. Let's illustrate the creative approach by rewriting our credit card test cases:

Table 1

Test case ID	Priority	Card Type	Expected Result One
CCPG0001	1	Visa	10
CCPG0002	1	MasterCard	20
AEPL0001	1	AmEx	30

SETUP and ADDITIONAL INFO

Go to Test Portal>More Stuff>QA KnowledgeBase to "See QA KB"

In order to run SQL1 go to Test Portal>Helpers>DB Connect Utility.

SQL1:

select result from cc_transactions where order_id = <order id>;

How to get credit card balance:

Test Portal>Helpers>Credit Card Balance Viewer

IDEA: payment can be made by credit cards from Table 1.

1. Generate Credit Card (See QA KB) and copy card number into the temporary text file.
2. Get credit card balance and write it down: _____
3. Create New Account (See QA KB)

4. Login
5. Find Search Keyword (See QA KB)
6. Do search
7. Add found book to the Shopping cart.
8. Copy card number from the temporary text file and Do Checkout (See QA KB) !!! While doing it, write down amount of payment: _____
9. Write down order id: _____
10. Make DB query: SQL1
- 10.A. Verify that
Step 10 result = Expected Result One
11. Get credit card balance and write it down: _____
- 11.A. Verify that
Step 11 result = Step 2 result - Step 8 result

Three Important Factors That Affect Test Cases

Each start-up is different, so you can be very creative in making your test cases as effective as possible, wherever you work. Let's look at three important factors that can affect both the formal and actual sides of your test cases:

1. Importance of the project
2. Complexity of the project
3. Stage of the project

1. IMPORTANCE OF THE PROJECT

Let's look at two hypothetical bugs. Please note that these bugs never occurred in reality; I made them up for the sake of illustration.

YouTube bug: 10% of the ending of the video clip is missing after an .avi file upload.
PayPal bug: 10% of an amount is missing after a bank-to-PayPal transfer.

I bet most of us would be somewhat annoyed in the first case and REALLY pissed off in the second case.

Here is another perspective:

- Can users sue YouTube for a faulty upload causing a loss in playback time? No.

- Can users sue PayPal for a faulty transfer causing a loss in money? Yes.

So, while YouTube is an amazing project, it deals with entertainment, and if it doesn't function it's upsetting, but you are just a click away from other similar Web sites.

It's totally different for PayPal, where people's finances are involved. If money is lost, a transfer is delayed, or an account is blocked, it can have really negative consequences for both the user and the company.

2. COMPLEXITY OF THE PROJECT

Some projects are super simple (ShareLane) and some projects are super complex (PayPal). Complexity is rooted in the technologies used, the number of lines of code, integration with vendors, etc. As a rule, projects of high importance are highly complex.

3. STAGE OF THE PROJECT

A project can be at an early stage when only a general idea and a simple prototype exist or at its maturity when hundreds of folks are employed and millions of users enjoy the product. A product in its early stage usually has low importance and low complexity.

Depending on these three factors, we develop our testing strategies, and these will also include the standards and requirements for the formal side of test cases. Today we have learned several good approaches to test case generation and management, but in many start-ups the above 3 factors will simplify test case related activities. As a rule, **during the first months of a start-up's existence** (i.e., when importance and complexity are low) **it's totally okay to have all or the majority of test cases in form of checklists.**

Checklists

Checklists are simple test cases that assume that the test case executor knows the application well, or can figure out how the application should work. As a rule, advanced verifications like DB queries are not included in checklists.

Example:

Action	Expected	Check
Do transaction with Visa	Success	<input type="checkbox"/>
Do transaction with MasterCard	Success	<input type="checkbox"/>
Do transaction with AmEx	Success	<input type="checkbox"/>

Before executing the test case, print out the checklist and put a checkmark if something works and no checkmark if something is broken or the execution is blocked.

For projects of low importance, it's okay to use checklists even after several years of existence.
For projects of high importance, highly formalized test cases are a must.

Lecture Recap

1. The test case is a piece of test documentation prepared and used by a tester to check for one or more expected results.
2. The test case steps (procedure) serve as an instruction of how to reach an actual result (output). Excessive detailing of steps causes difficulty in test case maintenance, while excessive abstraction causes difficulty in test case execution.
3. The steps for repetitive scenarios can be moved to a separate helper document inside QA Knowledge Base, so in the test case we simply refer to that document: e.g., "Do Checkout (see QA KB)."
4. The test case execution ends with a PASS or FAIL result. A FAIL result is desired, because in that case we have a bug (provided that the test case reflects what's really expected).
5. The test case execution is considered incomplete if the tester couldn't execute **all** the steps: e.g., situations where the test case execution is blocked because of a bug in the middle of the execution.
6. A test case should be independent from **all** other test cases; i.e., there should be no reliance on test cases from its own and/or any other test suites. A good way to check the independence of test cases inside a given test suite is to change the order in which the test cases are executed (i.e., if the test cases are truly independent, the order of execution doesn't matter).
7. The following test case attributes are extremely useful:
 - A **unique ID** is unique not only within a test suite, but also among **all** test cases existing in the company.
 - A **priority** is a value from 1 to 4 (inclusively) which reflects the importance of a test case. The practical consequence of having differences in priorities: *A P1 test case is more important than a P2 test case, and if we have time to execute only one of the two, we would execute the P1 test case.*
 - An **IDEA** – a written description (in human language) of a test case idea.
 - The **SETUP AND ADDITIONAL INFO** gives the test case executor helpful information for test case execution. This section is commonly used to store data, account information, SQL statements, and other pieces of information which help to execute test cases and make them more maintainable.
 - The **revision history** helps us to know history behind test case modifications.
8. Test case maintainability is about the ease and comfort with which we are able to modify a test case. Test case maintainability is one of the most important formal aspects to keep in mind during test case generation/modification.
9. Each test case checks only one testing idea, BUT two or more expected results are totally legitimate if we need to perform several verifications for that testing idea.
10. Below are examples of bad practices during test case generation/modification:
 - Dependency between test cases;
 - Poor description of the steps;

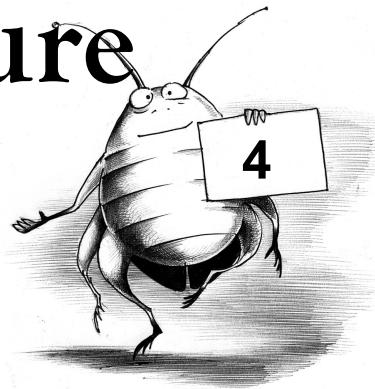
- Poor description of the IDEA and/or expected result.
11. Test cases are combined into test suites. As a rule, the whole test suite is placed into one file; e.g., a MS Word document.
12. As a rule, test suites include test cases which check
- the same area of the software (e.g., "Checkout");
 - items from the same spec.
13. It's a good practice to create a new test suite for each new spec. Even if the new test cases are written to test the same functional area (e.g., Credit Cards), it's better to create a temporary test suite with new test cases and then move survivors into the permanent test suite after first execution.
14. In the majority of situations, test cases are written after a simple reading of the spec, talking to developers, and taking other actions to understand what the **future** software is supposed to do. In other words, your familiarity with any future software AND its testing is purely theoretical. Thus, **test case modification during the first execution is always a reality**. Fresh test cases are not supposed to be perfect. After all, everyone knows what an abyss can exist between a theory and a practice.
15. It's imperative that during test case generation/modification we always think about the testers who will execute test cases after us.
16. Just like humans, test cases have the following states: "Created," "Modified," and "Retired." Unlike humans, test cases don't die. We want to keep them alive for a number of reasons; for instance, as a memory. So, if a certain test case is no longer applicable, we don't brutally delete it, but simply cut and paste it into the file with the same name as test suite file, but located in a special directory ("retired_testcases"). If the whole test suite cannot be used anymore, then we just move it into the same "retired_testcases" directory.
17. Be creative!!! And remember that
- Importance of the project
 - Complexity of the project
 - Stage of the project
- affect formal and actual side of test cases.
18. In this lecture we've covered only the formal side of test cases. It's important to understand that **the gist of a test case's existence is to help us find bug (-s)**. Bug finding techniques will be explained in great detail later on.

Questions & Exercises

1. Name one part of a test case without which the test case cannot exist.
2. Why do we need steps in a test case?
3. List the results of a test case execution. Which result is desirable for us as testers?
4. What shall we do if the test execution is blocked?

5. Why should a test case have only one IDEA?
6. List the useful attributes of a test case and substantiate why shall we love and foster each of those attributes.
7. Do we change the test case ID if we change the test case and/or move the test case to another test suite?
8. Develop your own way create a test case ID (e.g., the spec ID can be a part of the test case ID).
9. What is a data-driven test case? Is it harder or easier to maintain a test case after we convert it into a data-driven format?
10. Why does test case maintainability save time?
11. List bad practices during test case generation/modification. Why should we avoid such practices?
12. Why is it a good practice to create a separate test suite for each new spec?
13. Is it expected that the tester will revise his or her freshly baked test cases?
14. Which criteria can we use to put test cases into the same test suite?
15. List the attributes of the test suite header.
16. List the states of a test case.
17. Why is it not recommended to permanently delete test cases?
18. Do we have some kind of set-in-stone standard for the test case format?
19. What is the difference between a test case IDEA and an expected result?
20. Write a test case with one IDEA and two expected results. Use a situation from everyday life.

Lecture



The Software Development Life Cycle

Lecture 4. The Software Development Life Cycle.....	66
Quick Intro.....	67
Idea.....	68
Product Design.....	71
Coding.....	87
Testing And Bug Fixes.....	111
Release.....	113
The Big Picture Of The Cycle.....	134
Lecture Recap.....	137
Questions & Exercises.....	142

If everything seems under control,
you're just not going fast enough.
- Mario Andretti

Have no fear of perfection - you'll never reach it.
- Salvador Dali

Quick Intro

The Software Development Life Cycle - SDLC (we'll simply call it the "Cycle") **is a way to get**

- from an idea about a *desired software*
- to the release of the *actual software* and its maintenance.

The effectiveness of an Internet company and hence the happiness of its users fully depends on:

- how effectively each stage of the Cycle works and
- how effectively the stages interact with each other.

Today we'll cover a model of the Cycle called "Waterfall," which is used in 99% of all start-ups because it's the most straightforward way to develop software. (In fact, if you ask a programmer, "What model of SDLC do you have?" and he or she answers you "What?", then it's a Waterfall.)

We are going to go through each stage of the Cycle and will learn:

- the practical side of each stage,
- the relationship between different stages.

At the end of this LONG lecture, we'll look at a high-level overview of how the Cycle works (The Big Picture of the Cycle).

Don't worry if you don't understand all the material right away. We'll be covering A LOT of things, and some of them are not easy to grasp just hearing about them for the first time. You can:

- Read the material again
- Google it
- Send me an email with the subject line, "What the heck were you talking about?"

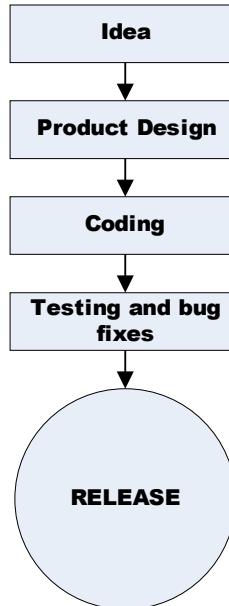
I'll try to minimize remarks like "In some companies it's called this or works this way, and in other companies it's called that and works that way." It's impossible to embrace such a variety, but if you grasp the main principles, you'll be able to easily connect what you've learned here with the reality of your present (or future) software company.

So, let's say "Hi" to

1. Idea
2. Product Design
3. Coding

4. Testing and bug fixes
5. Release
6. Maintenance

Here is a flowchart (Maintenance is not included intentionally, later on you'll find out why):



Idea

First let me tell you how software start-ups are often born. And please note that I'm not joking here (well, maybe a little bit).

Once upon a time in California...

One sleepy Sunday two friends, John and Paul, both programmers, meet at a sports bar in downtown San Francisco. Their conversation is about the two things that are always discussed in Silicon Valley:

1. How to create a cool start-up.
2. How to make enough money to retire young.

The bartender keeps the Budweisers coming, and nothing extraordinary is expected to happen, but suddenly Paul moves his head towards John, directs his pointer finger to the wooden ceiling of the bar, and says: "Got it!" That "Got it!" means the birth of another GRAND SOFTWARE IDEA, which sometimes can be as ridiculous as "Let's make a cool Web site to sell toilet paper." John says, "Wow!" and they start writing a

business plan. After a couple more beers, their business plan is finished and ready to be presented in front of venture capitalists (VCs).

If you have someone in front of you who you think might be a venture capitalist, you can ask him or her two questions:

Q1: Is your office located on Sand Hill Road in Menlo Park, CA?

Q2. Do you have 5 million dollars to invest in a cool Web site that sells toilet paper?

If the answer is Yes to both questions, the person is a venture capitalist.

Don't think that if someone wants to invest 5 million into some dumb project, he or she is stupid. Although venture capitalists are risk takers, they are smart risk takers, and they just happen to know that after they invest 5 million in a toilet-paper-selling Web site, somebody might pay 500 million to buy (or as they like to call it, "acquire") it.

BTW, John's Stanford pal Jerry happens to be a partner at a VC firm. The next day they all get together at Jerry's Sand Hill Road office. Guys pass their excitement onto Jerry's boss, and a week after that historic event the first round of financing (i.e., the first pile of money) in the amount of 5 million is available for wasting on the project of the century.

Paul, the guy who said, "Got it!" naturally becomes the CEO (Chief Executive Officer), and his buddy John becomes the CTO (Chief Technology Officer). Jerry becomes a member of the board of directors.

And now the craziness starts:

- The new company, **Just WOW Toilet Paper 4U, Inc.**, rents out an office as close as possible to the condo of the CEO (but not too far from the CTO's condo and Jerry's office).
- In order to buy Servers/Desktops/Linuxes/Oracles and hire inspired followers, money is pouring from their bank account like a Himalayan avalanche.
- Pepperoni pizza and Diet Coke become the everyday food and drink, even among the vegetarians and I-drink-only-water types.
- New lines of code appear with the frequency of bullets from a well-oiled AK47.
- Yahoo! Messenger becomes the primary means of communication between the spouses of the start-up employees and the start-up employees who just cannot afford the luxury of going home before 3:00 in the morning.
- Each atom of air inside the office is filled with excitement about the project and hopes about future fortunes.

- Nobody can convince start-up employees that their project is nonsense (but wait, maybe somebody will buy it!).

Got 500 mil for Just WOW Toilet Paper 4U, Inc?

As you were reading that story, you probably noticed that it all started with beer... sorry, with an **idea**. Everything starts with somebody's idea; software is not something unique in that respect. **First you envision, then you implement.**

Question: Who generates ideas at the software start-up?

Answer: As a rule, ideas come from the fellows who work in marketing or product management. But anyone at the company can come up with a great idea that can be implemented as a part of the software.

Example

In the case of YouTube, somebody came up with the idea to put thumbnails with related videos to the right of the main video clip. The beauty of that idea is that it makes site users want to watch more videos and thus spend more time on the site.

Question: What are other sources of ideas, besides the creative thinking of the start-up employees?

Answer: In many cases, ideas for new functionalities might come from:

- Users, e.g., via customer support people.
- New contracts, e.g., a company starts to work with another credit card processor.
- Internal data, e.g., by analyzing data from the database and log files, we can discover:
 - a. how users interact with our Web site (e.g., behavioral patterns) and
 - b. what social layers and types (e.g., age, education, gender, wealth, interests) our users represent.
- Cool features found on competitors' Web sites
- Other sources

When presenting The Big Picture of the Cycle at the end of this chapter, we'll refer to the person who generates ideas as the "marketing dude."

BTW

Here is a good way to collect ideas from within the company:

Create a special Web page on your Intranet called something like "Idea Vault." Everybody in the company should be able to read and edit it. This page is valuable because:

There are a lot of great ideas in the people's heads and there are a lot of great ideas being communicated during lunches, meetings, etc. But for many reasons (like the need to rationalize new ideas before you bring them up or because people just forget stuff), many great ideas stay in those heads and die in restaurants, meeting rooms, and other places. This page gives everyone an opportunity to express an idea in easy and nonobligatory way.

You'll be surprised how fast this page will be filled with great stuff, because start-up folks are very creative and LOVE to express themselves!

Two more things about an "Idea Vault" page:

- You should clearly communicate that a person should NOT think twice before putting his or her idea onto that page. That's the whole point. **"If you've got an idea, just go ahead and record it. DO IT even if you think that the idea might seem stupid, unexpected, or irrational. If you think it's a good idea, BRING IT ON!"**

- You should make sure that everyone in your start-up knows about the page. Send out an email with a link to the page with an explanation of why your company needs that page. I guarantee that everyone will love your initiative.

Product Design

As a rule, ideas are accumulated into a Marketing Requirements Document (MRD). The gist of the MRD is: **"It would be good to have these functionalities on our Web site."**

Then, ideas from the MRD are approved, disapproved, or ignored by the management, and those ideas that were approved are passed on to the product managers (PMs), who create specs based on those ideas.

Brain Positioning

As a rule, a "manager" is a person who manages other people.

Product managers don't manage people; instead, they manage products. The "product" is a large or small functional area of a Web site; e.g., at MySpace, there can be a product manager who specialized in the functional area called "communication between members." If the PM is a good one, his or her specs are well written and easy to use; if the PM is a bad one, his or her specs are misleading and bring more chaos than order.

Specs can also be called:

- the PRD (Product Requirements Document)
- the BRD (Business Requirements Document)
- simply the "requirements"

The gist of the specs is: **"These are the functionalities that we are going to implement on our Web site."**

We need specs to design the product. The difference between the idea and the design is this:

The idea is a **description of the purpose** (e.g., "We want to enable people to do this and that").

The design is a **description of the way to reach the purpose** (e.g., "In order to enable people to do this and that we have to do the following: ...").

The most effective PMs are those who have:

A. Professional and/or educational backgrounds in the area they specialize in (e.g., if someone is designing banking software, previous banking experience is very helpful).

B. A little bit of a technical knowledge; e.g., understanding the difference between the Web server, application core, and database (we'll learn about this soon).

Part A is simply needed to make sure that the PM knows what he or she is talking about. I wouldn't be happy about working on banking software with a PM who doesn't understand the difference between debit and credit cards (the first is needed to retrieve your own money, the second is needed to borrow money). Also, **a PM should be an educator**, because the specs should reflect the product design in a way that let developers and testers completely understand what they are required to develop and test.

Part B enables PMs to speak the same language as developers. I'm not talking about Chinese, English, Hindi, or Russian. I'm talking about the common technical language that's an essential part of communication inside a software company. It's a bit hard to work with a PM who doesn't know that "IE" stands for "Internet Explorer."

Each spec should have:

- a **title** (preferably unique); e.g., "Functional Improvements for the Search"
- a **unique ID**; e.g., #8765

Specs should be divided into indexed sections and subsections for ease of understanding and reference.

Each spec should have a priority. Usually this is a number from 1 to 4 (inclusive). The P1 spec is the most important. Here is why we need spec priorities:

- We can have a situation where the expected work to implement functionalities from two or more specs exceeds the availability of the engineering resources. For example, the developer and tester will not have enough time to implement and test all of those functionalities. In that case, the implementation of functionalities from specs with lower priorities (e.g., P2 is lower than P1) can be dropped from the coming release and rescheduled for one of the future releases.

- The developer and tester should always start their programming and testing with functionalities from the spec with the higher priority.

- The functionalities from the spec with a higher priority should be tested more thoroughly than those with a lower priority. We all know that it's impossible to fully test a more or less complex system, but we surely can (and must!) provide special treatment for those specs with higher priorities. That doesn't mean that P4 specs will be tested as though they mean nothing to the company; it simply means that we have to give more love to the testing of a P1 spec than we give to a P4 spec.

The spec priority is assigned to each spec by the PM's manager or by the PM himself/herself.

The Good, the Bad, and the Ugly Specs

Good specs, like good laws, have the following seven characteristics, or rules:

- | | |
|---------|--|
| Rule #1 | Clarity of details and definitions. |
| Rule #2 | No room for misinterpretation. |
| Rule #3 | Absence of internal/external conflicts. |
| Rule #4 | Solid, logical structure. |
| Rule #5 | Completeness. |
| Rule #6 | Compliance with laws. |
| Rule #7 | Compliance with business practices. |

Problems in specs show up if the PM has broken one or more of these seven rules.

Breaking Rule #1: CLARITY OF DETAILS AND DEFINITIONS

Example

Here is an extraction from spec #1111: "New User Registration":

"1.5. During New User Registration the system should check that the email address entered by the user has good characters and has a global domain name at the end of the address right after the '.' (dot)."

Does this sound clear to you?

What characters are “good”? What is the “global domain name”? What happens if “bad” characters are entered?

The PM must do **three things** to clarify what is meant by a valid email address:

1. Specify that the email address contains only one "@" (at) character.
2. Provide a reference to or a complete list of **illegal characters** for email addresses (in other words, characters that cannot be included in an email address; e.g., "," (comma)).
3. Provide **rules about the format of an email**; e.g., there should be at least one alphanumeric (a-z, 0-9) character before the @ character: a@sharelane.com.

Next, "global domain names" is not a conventional term. The correct term is "top-level domain." The PM should use the correct term and provide rules about top-lever domain formatting.

BTW

A top-level (or "first-level") domain has:

- either a generic name with three or more characters; e.g., "com" or "info"
- or a two-character territory code based on the standard ISO-3166; e.g., "ru" for Russia

Consequences

The standard practice of a new user registration consists of three stages:

1. The user submits the Web registration form.

2. The Web site sends the user an email with a confirmation link.
3. The user clicks the confirmation link, and his or her registration is complete.

So, if the system doesn't check email addresses for two or more "@" characters, then if the user erroneously submits a registration form with such an email address (e.g., anna_fuente@@sharelane.com), no error message will be displayed, and the user will never receive the email with the confirmation link. The user would wait for a while for an email from your site and then, being tired of waiting, he or she would just type in the URL of your competitor into his or her browser's address bar.

BTW

A URL (Uniform Resource Locator) is unique address of the resource (usually file) on the network; e.g., on your local company's network (Intranet) – or on network of networks (Internet). If URL describes location of the file

SL

- You can provide the filename directly; e.g., "http://www.sharelane.com/test_portal.html" or
 - in certain cases, the Web server would send you the needed file. For example, if you type "http://www.sharelane.com/index.html" or "http://www.sharelane.com", you'll get the same result.

Also, in case of Web pages, you don't need to type "http://" because the Web browser will add it automatically. That's why when somebody tells you URL, they usually start with "double-u double-u double-u dot."

Breaking Rule #2: NO ROOM FOR MISINTERPRETATION**Example**

There is a classic comedy act when two people are talking about different things, but don't understand this and keep going on.

Two spouses talking over the phone:

- *You know, honey, I'm so sick and tired of spam [“spam” meaning junk mail]*
 - *Too bad, my dear, I'm afraid that you'll be without dinner tonight.*
 - *Why?*
 - *Because we have nothing but Spam in our home [“spam” meaning canned meat]*
 - *So you want me to go hungry?*
 - *No, but you said that you are sick and tired of spam!*
- (to be continued)*

In the above example, this dialog happened only because "the honey" misinterpreted the word "spam" (not everyone is dealing with computers, you know). The "honey" is not to blame because "the dear" should've known that "the honey" might not understand the intended meaning of "spam" (junk email). In our everyday communication we have this kind of situation very often, and in some cases it ends up in an ugly way – like "we are not friends anymore" just because someone misinterpreted something

(words, gestures, etc.). Let's remember poor Romeo who misinterpreted Juliet's sleep as a death and finished off himself.

Consequences

If a spec opens the door for misinterpretation, there is the danger that the programmer and/or tester will misunderstand that spec, and thus the code and/or test cases won't reflect what the PM **really** meant while writing the spec. It sounds simple – "won't reflect" – but in reality this can have very serious consequences for the company, especially when big money and important contracts are involved.

The most obvious way to avoid misinterpretation of the specific terms used in the spec is to write down both the **terms** and their **clear definitions** in the beginning of the spec.

Later on, we'll talk about taking other measures to prevent spec misinterpretations.

Breaking Rule #3: ABSENCE OF INTERNAL/EXTERNAL CONFLICTS

breaking rule #3: absence of internal/external conflicts

Example

Spec #1: "8.1.1. For delivery, user can enter any postal address within lower 48 United States."

Spec #2: "7.3. Delivery must be made only to the billing address."

Consequences

If the conflict between these two specs is not sorted out, we can have the following situation:

- One programmer (following Spec #1) writes code to allow users to enter any address for delivery.
 - Another programmer (following Spec #2) writes code to prevent delivery to any address except the billing address (e.g., where the credit card is registered).

What can happen next is that during Checkout a user enters his or her mother's address to send her a Christmas gift, but his or her mother will never receive that gift because while the software allowed the user to enter any address, it prevented the dude in the warehouse to send the item, because the provided address didn't correspond with billing address.

Breaking Rule #4: SOLID, LOGICAL STRUCTURE

Example

*"Hey diddle, diddle,
The cat and the fiddle;
The cow jumped over the moon.
The little dog laughed to see such fun,
And the dish ran away with the spoon."*

Thanks to the unknown author of this nursery rhyme for illustrating how disconnected the components of the same “system” can be!

Consequences

Basically, the solid, logical structure of a spec is about the connections between topics, ideas, data, examples, etc. If the items of the spec are mixed and intermingled like spaghetti, this is a sure way to lead spec users (developers, testers, etc.) into confusion, or worse, into misunderstanding.

A confused person usually asks questions, while a person who misunderstood something might not be aware of it. Like Mark Twain said: "It ain't what you don't know that gets you into trouble. It's what you know for sure that just ain't so."

The golden rule for PMs should be: Write your spec like a tutorial, using clear, simple language and an easy-to-follow structure.

Breaking Rule #5: COMPLETENESS

Example

In trying to fight credit card fraud, credit card companies introduced a new safety measure called CVV2 (Card Verification Value 2). CVV2 is 3 (all cards except AmEx) or 4 (AmEx only) digits following the card number on the back of the credit card. If for whatever reason the PM failed to include the CVV2 verification code into the spec, the company could get exposed to fraudsters, who usually have all the credit card information except the CVV2 numbers.

Consequences

Many Internet companies, including several online payment systems, ran out of money and died because of Internet fraud. In our age, issues about security cannot be taken lightly, and the failure to acknowledge the importance of fraud prevention will backfire pretty quickly and severely. Even if your company can deal with the enormous losses caused by Internet fraud, the company's customer support and financial and legal departments will have A LOT of trouble trying to sort out the consequences of security related mistakes.

Bottom line: "**Things omitted are often more deadly than errors committed**" (Leo F. Buscaglia).

Breaking Rule #6: COMPLIANCE WITH LAWS

Example

Because of an embargo, Cuban cigars cannot be sold in the U.S. So, if somebody sets up a Web site that sells Cuban cigars to customers in the U.S., that person is breaking the law.

Consequences

Remember Napster...

Breaking Rule #7: COMPLIANCE WITH BUSINESS PRACTICES

Example

Because of the many variables during a money transfer, banks cannot guarantee the exact date when a money transfer will be complete. That's why it's business practice to inform customers about the **approximate** number of days in the form of a range: e.g., "three-to-six business days."

So, if a money transfer usually takes three to six business days, we should not promise our customers that they will get their money on an **exact** day. It's much better to just have a nice, simple confirmation message: "Money transfers usually take three-to-six business days."

Consequences

Dr. Rodionoff is sure that on the exact date promised by our company his account will have 300.000 pounds of sterling. He flies to London to participate in an auction of Russian art of the nineteenth century to acquire the dream of his life: a beautiful landscape painting by Ivan Aivazovsky. He bids aggressively and wins the painting for 235.000 pounds. Now it's time to pay. The cashier takes his debit card, and after several futile attempts she says out loud (in the way only cashiers can do) that Dr. Rodionoff's card has been declined.

First question:

Will Dr. Rodionoff ever use our services again? No.

Second question:

Will Dr. Rodionoff ever recommend our company to anyone except his enemies? No.

Third question:

Which words and expressions will Dr. Rodionoff use from now on if somebody mentions our company? Let's not elaborate on that.

The last thing I'd like to mention about incorrect specs is this: **Some PMs are confident that their specs will provide instructions about purely technical aspects to programmers**, like setting up the relationships between tables in the DB or the names of variables to be used in the code. Generally, this is a BAD idea, because even if the PM has a programming background, he or she has zero to vague understanding about the reality of coding in that particular company. Why? Because it takes full-time involvement and practical coding experience to understand what the heck is going on in the company's programming scene. So if those technical details are provided in the spec, the tester might write the wrong test cases, because the programmer will most likely ignore PMs poking into the programmer's business.

If a PM in your company doesn't understand all negative consequences of that wicked approach, ask him or her to write software code instead of writing specs. I bet your PM would prefer to write specs!

Specs have the following status order:

1. During the spec writing, its status is "Draft."

What's going on: the PM is working on the spec.

2. After the spec is finished but before it's approved, its status is "Approval Pending."

What's going on: After the spec is written, the PM sets up a spec review meeting with the programmers and testers. Sometimes, the spec is just emailed to those programmers and testers who are going to work with it.

3. After approval, the spec's status is "Approved" or "Final."

What's going on: If the spec is accepted by the meeting participants, or if the recipients of the email with the spec have sent positive feedback to the PM, the approved spec is immediately uploaded to the company's Intranet – available to all those who need to see it. If the spec is not approved, it all goes back to Step 1.

Let's proceed.

After his famous game of chess in 1972 with Bobby Fischer, Boris Spassky was probably thinking, "I should've smashed his damn knight with my queen." However, there was nothing that he could do about his loss because it was in the past, and the past is irreversible. But for **some** PMs, a spec is not a matter of the past; it's like clay which shape can be changed... anytime. So that PM can wake up in the morning

- with fresh idea about improvement for an already approved spec
- with the realization about some omission in the approved spec

After coming into the office, he or she can edit the spec and upload it back to the Intranet server. The next morning everything can start over: an idea about editing, editing, uploading... The next morning... (we'll stop right here).

This is what happens next:

Usually programmers and testers print out the spec in order to start working on the functionality described there. Naturally, the spec is being printed out at different times, and if the spec was **secretly** changed between printings, several people might have several different editions of the spec.

Even if developers* and testers print out the same edition, in the case of the following concealed spec modification, their work might be wasted because they used old spec edition.

*"Programmer" and "developer" are used interchangeably in this Course.

The keyword here is "**secretly**," and the described situation is about **ethics**, not about necessity of setting a total ban on modifications of an approved spec. After all, we are imperfect humans and new bright ideas can come to us after "the ship has left the harbor".

In many cases, the PM is asked to change an approved spec by the management.

Example

Here is the situation:

On the morning of November 25, the PM's manager sends an email to the PM with an urgent request to change spec #8337.

One day before (November 24):

- The PM was writing spec #9211 (spec #8337 had been finished and approved several weeks ago);
- The developer was writing code for spec #8222 (the code for spec #8337 has already been finished);
- The tester was writing test cases for spec #8222 (test cases for spec #8337 have already been created);

On the next day (November 25):

Spec #8337 itself, as well as its code and test cases must be changed; i.e., at least three persons must:

- drop their current (and maybe important!) tasks;
- remember spec #8337 and understand the changes;
- spend time for creating modifications of their work

This situation is an ideal **environment to bring software bugs to life**, because **it involves interrupted people who just don't have enough room in their schedule and in their minds to fully dive into all the nuances and realize all the risks which those modifications can introduce**. Furthermore, **new projects** might be

- implemented with less care because of having less time to spend on them;
- excluded from the coming release because of not having enough time to implement them.

So what can we do to prevent all that jazz with changing specs? Well, if management **reasonably** believes that the spec must be changed and/or the PM "remembered" the really really important thing, then there's nothing we can do – the spec must be changed. That's a bad, but sometimes inevitable situation, and the person who requires the spec change better have a very solid argumentation for it.

But on the bright side, there IS something that we can do regarding two unpleasant spec changing situations:

- A spec is secretly changed.
- The spec changes are not approved by programmer and tester.

And our answer is: **The approved spec edition must be "frozen."**

Any Internet company has software to control file versions. In the majority of start-ups, it's the old, free, faithful CVS (Concurrent Version System).

Using the CVS, it's possible to lock a directory so none of the documents in it can be modified.

Here is an example of the spec freeze process:

1. By the specified date all specs for the concrete release must be approved. If the spec is not approved, it's dropped. See you later, alligator.
2. The CVS directory with the approved specs is locked, and nobody can make any changes to those specs, unless the **Spec Change Procedure** is followed.

BTW

The CVS set up and any other technical aspects of a spec freeze are usually provided by the release engineers (RE).

The spec change procedure usually has these stages:

1. Approval of all changes by the same people who approved the original edition of the spec.
2. Opening the CVS directory to save the new spec edition, then locking that directory again.
3. Sending an email with a list of changes and the names of the people who approved those changes.

If it's a start-up, then it makes sense to send this email to all developers, testers, and PMs. The reason for this wide distribution is simple: in many cases a PM might not know who already is using previous edition of the spec.

Remember, it's not possible to live without changes in the approved specs, but by a spec freeze, the introducing the Spec Change Procedure, and a thorough audit of the necessity of each change (with the participation of the management), we can prevent a lot of serious software quality issues.

BTW

It's also a good idea to turn on text editor functionality "Track Changes" every time when approved spec is being changed. In that case, it's possible to see changes without comparing different versions of file with spec.

Brain positioning

Having rules in place does NOT mean that these rules are being followed. Anyone who has kids knows what I'm talking about. So the idea is not only to have rules, but also to ENFORCE those rules. I'm not talking about smashing keyboards against the culprits' heads; there are much more civilized methods like, "Sorry dude, no bonus for you this quarter."

Another important point: *If the management supports the introduction of rules, but doesn't enforce them, those rules have no meaning.* How do you make the management get serious about enforcing the rules? There is no universal recipe. The management usually wakes up after a couple of postponed releases and ugly P1 bugs released to customers.

The rules must be concrete and clear so that folks have no problem understanding them.

Talk to the PM manager about introducing and enforcing **concrete, clear rules on spec writing and the modification of approved specs.**

Let's proceed.

As you already know, one of the serious problems associated with specs is called **misinterpretation**; i.e., a situation that occurs when the spec audience (mainly programmers and testers) misunderstand the PM's words. What's dangerous about this is that folks who misinterpret something usually:

- are 100% sure they have a clear understanding of that something;
- don't ask questions, because why ask questions about something when that something is obvious?

Specs are often misinterpreted because of the PM's failure to **present things from different angles**. So, it's very important for PMs to **illustrate** their specs. Below are the three most common ways to do that:

- **Examples**
- **Flowcharts**
- **Mock-ups**

Brain positioning

Illustrations are beneficial for both parties:

- the author who tries to illustrate his or her ideas gains a better understanding of those ideas;
- the reader who receives the illustrated ideas has a better chance to fully comprehend whatever the author is trying to say.

Using **examples** is the most universal and powerful way to explain things. Two ways to create an example are:

- Bring a direct situation to the attention of your audience. For example, when we talked about log files we saw an example of an actual log file (Test Portal>Logs>cc_transaction_log.txt).
- Take a concept and create an analogy. For example, remember the story of my friend who thinks about his soul/body as a software/hardware. An analogy is a marvelous way to create an example, because **you can present things from an unexpected perspective and thus invoke another path of thinking for yourself and your audience**.

Now, let's talk about **mock-ups**. Folk wisdom tells us, "**A picture is worth 1000 words**," and "**Seeing is believing.**" For this reason, it's a very good idea to attach mock-ups of the UI (User Interface) to the spec. The process of creating mock-ups is pretty simple:

1. During (or after) creating a spec, the PM takes the Web page generator (like Microsoft FrontPage) and by simple manipulation creates a prototype of a Web page with buttons, fields, images, and other details of the UI.
2. Then this Web page
 - is published on the Intranet, and a link to it is provided in the spec *and/or*
 - the Web page design is saved as an image (e.g., in a jpeg format), and that image is inserted into the file (e.g., a Microsoft Word file) with the spec.

It's also a good idea to print out those mock-ups and attach them to the wall in the office.

Example

Here is a description of the UI from the spec #1111, "New User Registration":

- Page 1 has a field for "ZIP code" and a button "Continue"
- Page 2 has fields for "First Name," "Last Name," "Email," "Password," "Confirm Password," and a button "Register"
- Page 3 has a confirmation message.

All fields are required, so if user enters an invalid or null value for any field, the system generates the same page, but with an error message. (Note that for the sake of this example we are not going to elaborate on what constitutes a valid input for each field, but in reality the PM must specify those details).

So:

- The PM creates three mock-ups for Web pages.
- Next he or she attaches them to the spec as pictures.
- Finally he or she prints them out and tapes them to the wall in this order:
Page 1-> Page 2-> Page 3

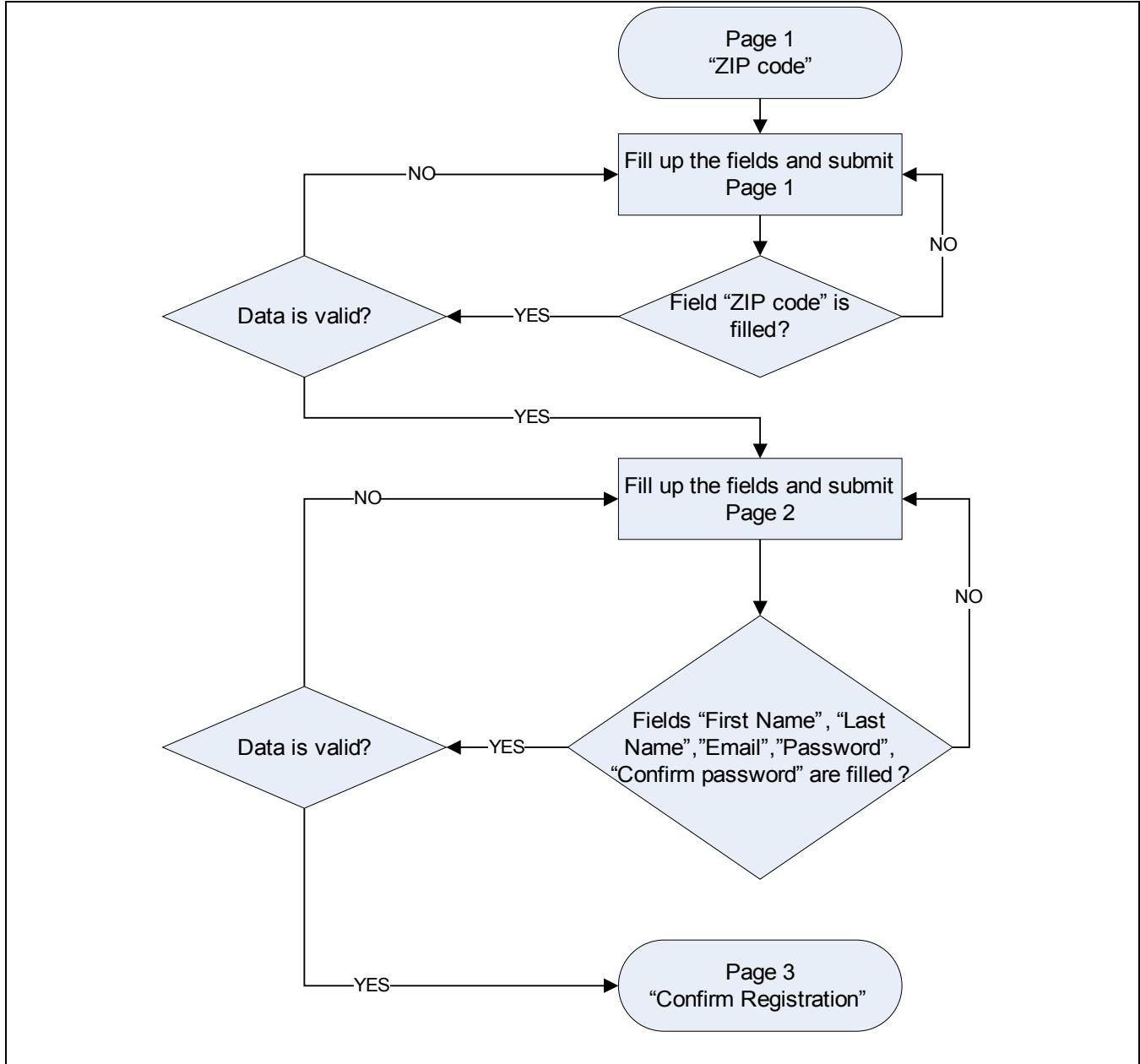
BTW

Mock-ups can have different degrees of abstraction; it's okay not to include elements of the interface that might have no relation to the spec. For example, in the case of mock-ups for registration we might not be interested in seeing the images on the Web pages.

The problem with mock-ups is that their purpose is to show user interface. Mock-ups usually give little clue about the logic behind that interface; i.e., about the algorithm of the program. To fill that gap, we use **process flowcharts**.

Example

Here is the registration in the form of a process flowchart (see next page):

**BTW**

Flowcharts can be created by both the PM and the tester. But no matter who the author of a flowchart is, it's usually a good idea to include it in the GLOBAL SETUP and ADDITIONAL INFO section of the test suite.

BTW

Microsoft Visio is a professional tool to generate flowcharts. You can download a trial version of it – just Google it. You can also create flowcharts using Microsoft Word.

Examples, mock-ups and flowcharts (let's call these sisters "**EMF**") help to:

- prevent software bugs,
- find bugs in the spec

by the following ways:

- The EMF is a description of the subject from different angles that helps different people to better comprehend the subject.
- The EMF creation is a process of rethinking, and rethinking leads to finding errors, i.e., bugs in the spec.
- Mock-ups and flowcharts are visual presentations, and in many cases they allow us to literally see spec errors with our eyes.

Since you've heard about mock-ups, but haven't seen them yet (the purpose of this was to let you feel the difference between **reading specs** and **seeing specs**), let me introduce you to the mock-ups for spec #1111, "New User Registration"

Mock-up of Page 1

A simple form with a light green background. It contains a text input field labeled "ZIP code*" and a "Continue" button below it. A small note at the bottom left says "*required".

Mock-up of Page 2

A form with a light green background, containing fields for "First Name*", "Last Name", "Email*", "Password*", and "Confirm Password*". Each field has a corresponding input box. A "Register" button is located at the bottom right, and a note at the bottom left says "*required".

Mock-up of Page 3

A small message box with a light green background containing the text "Click [here](#) to return to homepage and login".

Bonus: Mock-up of Page 2 in case of invalid data for email field.

Oops, error on page. Some of your fields have invalid data or email was previously used

First Name*	Max
Last Name	Kostas
Email*	max_kostas#sharelane.cc
Password*	
Confirm Password*	
<input type="button" value="Register"/>	
*required	

You can see all these mock-ups in action if you try to register on www.sharelane.com!

BTW

The mock-up of Page 2 and the bonus mock-up of Page 2 contradict the instructions found in the textual part of the spec:

- Textual part of the spec: "Last Name" is required field.
- Mock-up: There is no indication (asterisk "*") that the "Last Name" field is required.

Congratulations! We have just found a typical spec bug. And we also saw another illustration how internal conflicts within specs can produce bugs.

Brain Positioning

If you find a spec bug (remember that EMFs are parts of the spec), you have to file a bug report that requires the PM to fix the problem. In our case, either the textual part must be fixed to be in accord with the mock-ups or vice versa.

BTW, here is what I'd recommend for the bug report summary: "**Spec1111: internal spec conflict: not clear if "Last Name" is required.**"

Note that we put the spec ID at the beginning of the bug summary. Later on you'll find out why this is a good practice.

To conclude our short introduction to specs, I'll leave you with a good idea.

Each software company usually has an Intranet, an internal network for company workers only. The Intranet usually has a Web site for internal information sharing where insiders can:

- *read about the ethical standards of the company;*
- *see pictures from the last party when somebody drank too much beer;*
- *view instructions of how to set up a wireless access;*
- *read scary stories about the unfortunate destinies of PMs who secretly change approved specs;*
- *see contact info of each employee;*
- *read/see/view/create other useful stuff.*

So it's a good idea to publish ALL specs (from the past and the present) on this Web site. The easiest way to do it is to install some free Wiki program, like MediaWiki (<http://www.mediawiki.org>) on this Web site. With MediaWiki, a PM can easily create a Web page for every release and place there links to the files with the specs.

BTW

There is a free macro called Word2MediaWikiPlus (Google it) that you can use to convert documents from Microsoft Word (with tables, fancy formatting, etc.) to the MediaWiki format. The benefits of converting specs to the Microsoft Word documents into the Wiki format are these:

- We can use an internal MediaWiki search to search inside all converted specs.
- Some developers (e.g., Linux purists) refuse to install Microsoft Word on general principle.

In the future, when I refer to "Wiki" I'll be referring to an **internal company Wiki**.

Brain Positioning

Don't be shy to report bugs that you find in the specs. If your PM doesn't understand why you are doing that, explain to him or her that

- bugs seeded in specs can be transferred into the code and test cases;
- bugs found earlier cost much less than bugs found later (we'll talk about it in a minute);
- finding and addressing bugs found both in the code and the specs is not your choice, it's your responsibility.**

If your PM has a problem with that even after you explain the three points above, redirect him or her to your manager, and let your manager deal with the situation.

Brain positioning

As you already know, specs, and sometimes ideas for specs, can be buggy. That's why it's always a good idea for testers to attend product discussion meetings. **The sooner a tester is involved in the Cycle, the better it is for the software.** If the PM tells a tester that production discussion meetings are none of tester's business, that tester should have a really serious talk with that PM's manager. The tester's profession is somewhat unique, because **quality really depends on EACH participant of the Cycle**; in other words, **quality is a consequence of a joint effort**. So, when a tester gets involved in stages other than "Testing and bug fixes," it's a totally natural thing.

BUT, even if a test engineer is actively involved in broader quality topics, the major part of his or her work is to find bugs in the **software code** and address those bugs. As you know, bugs can come from

specs, and it's great to prevent/detect bugs before the "Coding" stage! But **the product we deliver to users is not specs, but a software code, and that's why the code to be released is the main focus of tester's attention.**

The next stage of the Cycle is Coding. Coding is performed by developers **while testers write test cases to check the code.**

Coding

In a nutshell, the programmer's job is to translate ideas from the specs (or from other sources, like a brainstorming session during lunch) into the software code.

The translation can be made:

- *either* directly (i.e., the programmer takes a spec and immediately starts coding (BAD idea)
- *or* after creating the software design documentation. The software design documentation is a detailed technical description about how the spec requirements are going to be implemented into the software code (GOOD idea)

Two common types of software design documentation are:

- 1. System/Architecture Design Document.
- 2. Code Design Document.

An emerging culture of the creation and maintenance of software design documentation is the first sign that a start-up is converting from a bunch of dudes who write code into a serious software company.

Brain positioning

"There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

– C.A.R. Hoare, computer scientist

Let's proceed.

In an ideal case, each developer has his or her own environment (i.e., software/hardware combo) with a certain version of the Web site. This is called the "development environment", "development playground", "playground" or "sandbox". The architecture of the playground imitates the architecture of the production environment. Here is a typical architecture of the production environment of Web-based application:

- Web server
- Application core
- Database

Let's take a quick look at how these components interact with each other.

BTW**SL**

You can look inside each and every Python script mentioned below if you go to Test Portal>Application>Source Code.

You can see HTML code of any Web page if you click

- View>Source from the menu of **Internet Explorer** or
- View>Page Source from the menu of **Firefox**.

SL

1. After the user enters "www.sharelane.com" into the address bar of the Web browser and presses the Web browser button "GO," request to retrieve the default Web page of www.sharelane.com hits the **Web server** of sharelane.com, and in response the **Web server** sends this to the hard disk of the user:

- a text file **index.html** (<http://www.sharelane.com/index.html>), containing HTML code
and
- an image file **logo.jpg** (<http://www.sharelane.com/images/logo.jpg>).

BTW

The first (default) page that a user sees after requesting the main Web site URL (e.g., "www.sharelane.com") is called the **homepage**.

BTW

The communication between the Web browser and the Web server is based on the request/response protocol (set of rules) called HTTP (Hyper Text Transfer Protocol). The currents of HTTP-based requests and responses are called **HTTP traffic** (or simply "Web traffic").

2. After the user clicks link "Enter", HTTP request is sent to the **Web server** to launch Python script **main.py** (<http://www.sharelane.com/cgi-bin/main.py>).

Python script **main.py** belongs to the **application core**. That script generates HTML code that is sent to user's hard disk via **Web server**.

3. After the user clicks link "Sign up", HTTP request is sent to the **Web server** to launch Python script **register.py** (<http://www.sharelane.com/cgi-bin/register.py>).

Python script **register.py** also belongs to the **application core**. That script generates HTML code that is sent to user's hard disk via **Web server**. HTML code generated by **register.py** includes Web form with text field "ZIP code" and button "Continue".

4. After the user presses button "Continue", form data (name of the field: **zip_code** and its value, i.e., user's input into text field "ZIP code") is passed through the **Web server** to register.py for **processing**.

5. If supplied ZIP code has correct format (5 digits), register.py generates HTML page with web form containing fields "First Name," "Last Name," "Email," "Password," "Confirm Password," and a button "Register". As we already know, that Web page is sent to user's hard disk by the **Web server**.

6. After the user filled all fields, he or she presses button "Register" and request containing Web form data is passed through the **Web server** to register.py for **processing**.

Script register.py does processing and if everything is OK (e.g., email format is valid), it inserts new row into DB table users (Test Portal>DB>Data>users) and generates confirmation Web page about successful registration. As we already know, that confirmation Web page is sent to user's hard disk by the **Web server**.

Brain positioning

The application core consists of files designed to:

- take input,
- process that input,
- deliver an output.

BTW

The term “application” (or “software application”) can be used in either a broad or a narrow sense.

- In the broad sense, it refers to the Web site in general.
- In the narrow sense, it refers to the concrete piece of software (e.g., Python script register.py).

We'll use the term “application” in its broad sense.

Brain Positioning

The **application core** is the most sophisticated part of the software application.

Here is an analogy:

When I visit the fine cigar store:

1. I ask a salesperson to bring me a box of cigars.
2. THE SALESPERSON ASKS ME MY AGE BECAUSE IT'S REQUIRED BY LAW.
3. IF AND WHEN MY AGE IS VERIFIED, I give the salesperson my credit card.
4. THE SALESPERSON PROCESSES THAT CREDIT CARD.
5. IF MY CARD IS NOT DECLINED.
6. THE SALESPERSON GOES TO THE BACK ROOM.
7. AND BRINGS ME MY FINE CIGARS.

All CAPS represent actions connected to the **processing** of my order (**application core**).

Can I get the box without having the salesperson's ears (**interface**) to receive an order? **No.**

Can I get the box without having that box in storage (**database**)? **No.**

Both the interface and the database are **essential** to our application, but again, the **application core is the most sophisticated and logically charged part**.

Programmers are primarily responsible for developing the code for the application core.*

**At start-ups the same programmers usually do it all: write code for the user interface and for the application core. Later, the company usually hires a special person, a UI developer, whose only focus is UI coding.*

Later on that code will be given to us, the merciless testers, who know perfectly well all the main reasons why bugs appear in the software code:

a. Bad and/or changing specs

We just covered this.

b. Personality of developer

For example, a developer might be irresponsible, not caring, or just lazy.

c. Not enough programming experience

A developer can be a responsible person, but has no idea how to do programming right.

d. Neglecting coding standards

We'll cover this next.

e. Complexity of the software

Many Internet projects are so complex that the brain of a mere mortal just cannot possibly predict all the consequences of the creation/modification/removal of a code.

f. Bugs in third party software

For example:

- bugs in operating systems;
- bugs in compilers and interpreters (the programs that translate software code written by humans into software code understandable by hardware; i.e., into 0's and 1's - we'll cover this in a minute);
- bugs in Web servers;
- bugs in databases;
- bugs in software libraries.

g. The absence of unit testing

Unit testing is testing performed by the programmer himself/herself. (The typical developer's excuses: "Why should I look for bugs while we have testers?" and "I don't have time."). We'll go over this.

h. Unrealistic time frames given to write code

We'll go over this.

Below are the measures to help us enhance programming practices and **prevent** a substantial number of bugs:

- 1. Hire good people, and be prepared to give them a break.**
- 2. Direct, fast, effective communication between coworkers.**
- 3. Code inspections.**
- 4. Coding standards.**
- 5. Realistic schedules.**
- 6. Availability of documentation.**
- 7. Rules about unit testing.**
- 8. "If it ain't broke, don't fix it."**
- 9. Being loved by the company.**
- 10. Having "quality" and "happiness of users" as fundamental principles of the company philosophy.**

Details:

1. HIRE GOOD PEOPLE, AND BE PREPARED TO GIVE THEM A BREAK

Hiring is a hard process because an interviewer communicates with a candidate for a maybe an hour or less and afterwards has to make a decision about both that candidate's technical skills and personal qualities. The simplest and very effective determination about a candidate's personality is this: **that person must be excited about your start-up**. If he or she is not, don't hire him or her. The reason is simple: **The start-up environment is about a gang of passionate dreamers who are sincerely excited about their product and willing to sacrifice their talents, time and efforts for the company**. You don't need sour, unmotivated folks in your team. *It's better to hire a motivated passionate beginner with the shine of inspiration in his or her eyes rather than a highly skilled jerk who will later contaminate the start-up atmosphere with his or her negative attitude.*

Here is another thing. A new programmer in the company often doesn't have enough time to get up to speed. As a consequence, he or she can write buggy code and/or screw up code written by others. That's why it does make sense to:

- give time and tutoring to new developers;
- give new developers easy projects to start with.

2. DIRECT, FAST, EFFECTIVE COMMUNICATION BETWEEN COWORKERS

It's very important to have this unwritten rule in any start-up:

"If you are asked for help, go ahead and help."

Yes, there ARE situations when it's not possible to interrupt your work; in this case, promise to get back to the person (**and really get back!**) or refer the person to someone else who *can* help. BUT in the majority of cases, it IS possible to spend several minutes with whoever asks you for help.

BTW

As a rule, a start-up is located in the same building and on the same floor, so in many cases people prefer to approach each other personally (rather than sending an email or instant message, or making a phone call). A very simple and effective way to notify others that you are busy and cannot respond NOW is to stretch some tape across entrance to your cubicle and attach to the tape a sheet of paper with these words: "Extremely busy. Email me." It's easy to do, but this can save a lot of time and concentration when you are working on something urgent.

Now, let's talk about what happens if people just ignore help requests.

Example

A programmer comes to the PM for an explanation about some part of the spec. The PM says: "Let's do it tomorrow, okay?" Okay. What might happen after a couple of "Let's do it tomorrow" without doing it tomorrow? Right! The programmer might start writing code even if he or she has doubts about the unexplained part of the spec. Needless to say, there is a good chance that code will not reflect what the PM tried to express.

Who is to blame in this situation? Actually, both the PM and the programmer.

The PM is to blame for failing to do his or her job properly.

The programmer is to blame because he or she:

- should have insisted on getting an answer from the PM;
- should not have done coding without a comprehensive understanding of the spec.

Here is a great quote from movie *Ronin*: "**Either you're part of the problem or you're part of the solution – or you're just part of the landscape.**" So, let's toast to being a part of the solution when we are asked for help!

BTW

A couple of other things about questions and answers:

- The programmer (like all other participants of the Cycle) should not be shy about asking questions (as many as needed!).
- In many cases it's a good idea to confirm your understanding of a verbal answer by email; for example, "Hey Mike, I just wanted to double check that I understood correctly item 12.2. of spec #9571. What you said was: '<explanations from Mike>.'" Why send such an email?

1. You'll show another party how you understood and digested information. If you didn't correctly understand the explanations about the spec, then the PM **must** respond with corrections.

2. You'll have an actual document (email message) to cover your ass, if needed. Actual email is much better as evidence than reference to some verbal blah-blah-blah. If you got the explanations about the spec wrong and sent an email asking for confirmation, but the PM didn't correct you (or gave you wrong information), then the **PM will be responsible for the consequent problems, not you.**

Management must understand that it's very important to help workers develop relationships with each other. One of the ways to do this is to have team-building activities. In my opinion, it's much better to go hiking or play paintball once a month rather than get together every Friday and marinate everyone's brains with alcohol. But, well, whatever works.

A technical aspect about communications is that each participant of the Cycle must be available for contact. Encourage everyone to publish their

- office, mobile, and home phone numbers
- work and personal email addresses
- instant messaging info

on the Wiki (you can find examples of these documents under Contacts in the Downloads section of qatutor.com).

BTW

It's also very important to agree on a time (e.g., four hours during the day) when everyone (both those who left at 6 pm and those who left at 3 am) must be in the office. I think that the time frame from 2:00 pm to 6:00 pm is just right. By 2:00 pm

- You'll be back from your lunch if you left at 6:00 pm previous day.
- You'll be back in the office if you left at 3:00 am the same day.

3. CODE INSPECTIONS

Some programmers have this concept: "*If I write code that only I can understand, then nobody will fire me.*" **WRONG!**

First, anyone can be fired, including CEOs (thanks to corporate corruption in the U.S., we have many stories to illustrate this point).

Second, this approach is simply dishonest, because nobody forced you to work at the company; it was your decision to work there. So, **if you agreed to work for the company, go ahead and do your best or leave if you don't like it there***.

* *If you are a consultant, then you usually have a specified period of time in your contract, i.e., three months, during which you must provide your services to the company. If you are an employee, then you're usually expected to give two weeks notice before you can leave. Read all documents before signing!*

To prevent “only-I-can-understand-my-code” behavior (which not only creates bugs, but also makes it harder to fix them), the company must have a **practice for code inspections**. This can be a weekly

meeting when the manager of the programmers prints out the code of a random programmer and the latter explains his or her code to his or her colleagues. I doubt that the programmer would have an "only I-can-understand-my-code" approach if he or she would most likely be asked at the meeting: "What the heck is wrong with your code, comrade?"

4. CODING STANDARDS

Code inspections have a great ally called "coding standards."

Remember the story about building the Tower of Babylon, when everyone started to speak different languages? The consequences were really sad: people on earth got disconnected by a language barrier. So in order to communicate, we need to

- learn each other's native tongue (very difficult if you have friends in many countries) *and/or*
- get a translator (costly and inconvenient for everyday use) *and/or*
- create some kind of **standard** (good idea!)

A similar mess like in Babylon happens at a software company if each programmer uses his or her own coding standard.

Example

Let's say that there is no naming convention (standard) for Python functions at our company.

IF

- Matt likes to name his functions using this format: "credit_card" (using lower case for all letters and an underscore between words) while
 - Anna likes to name her functions using this format: "CreditCard" (upper case for first letters of each word and no separator between words)

THEN

- it's possible that Anna (remembering her habits, but not knowing Matt's habits) would try to call Matt's function "credit_card" as "CreditCard." As a result, Anna's code wouldn't work, because Python doesn't care about Matt, Anna, Babylon or our super-duper Web site, but it does care about the precise name of the function when that function is being called.

Coding standards can include:

- rules about comments
- naming conventions for classes, functions and variables
- rules about formatting
- and a HUGE PILE of other things

Needless to say, coding standards must be published on the company's Wiki site, and every new developer must be referred to them. Coding standards must be adopted and followed in any software

company that has more than one programmer. For a new company, it makes sense to start with a simple version of coding standards and then update them as the company grows.

5. REALISTIC SCHEDULES

When I was a child, my mother used to ask me to help her sort rice. As wild as it might sound nowadays, rice sold in the Soviet Union was not edible until you spent several hours sorting it before cooking. Even after I became quite proficient, there still was a speed limitation, after which I'd start missing things like little stones, so my family members would have to spit out my omissions during the dinner. Why am I bringing this up now? **Every person has a limit, whether it's about sorting rice or writing a software code.**

If you create an unrealistic schedule, then either

- you won't get what you wanted OR
- you'll get what you wanted, but the result will *most likely* have poor quality.

This is exactly what often happens at software start-ups: developers are pressed to finish writing code in time, and while they do manage to produce the code on schedule, its quality might be really bad. Can we blame the developers? Of course not! We are all in the game called "start-up craziness" and unrealistic schedules are a part of our lives. **Timing is everything, and we have to release our product and win users before our competitors do.** That's why the tester's role in a start-up is critical – we allow the developers to move on to new projects while we are testing their OFTEN MESSY code. Like it or not, that's the reality of start-up life.

The good news is that there are ways to ease up on the pressure of unrealistic schedules. One of the most effective solutions is this: *before including spec into release, the engineering manager should ask the developer for a time estimate for coding that spec; i.e., how much time the developer thinks it will take to write the code.* Having estimates from all the developers, the management can:

- rearrange projects between developers
- drop specs with less priority
- ask the PM to exclude some functionalities from the spec

Our only "hope" is that one day our start-up will become a big company, and everybody will be able to leave work at 6:00 pm. But, trust me, you'll be missing those old good times when you were sleeping four hours a night but felt happy like never before!

6. AVAILABILITY OF DOCUMENTATION

ALL documents which are relevant to the Cycle, including the specs, spec change procedures, coding standards, test suites, etc.:

- MUST be available on the Intranet (e.g., by using Wiki)
- MUST be optimized for easy searching and browsing

It sounds simple and obvious, but if you don't think about it, others will have to waste time and effort to access your documents.

7. RULES ABOUT UNIT TESTING

Unit testing is a test performed by the programmer against his or her own freshly baked code. I must stress two points here:

1. It's very important to correctly create and introduce rules about unit testing. If you do it wrong, the developers will be really annoyed, because they are not paid for testing – we, the testers, are.
2. If we don't have concrete rules about unit testing, then the cost of the bugs will be increased.

Brain Positioning

The term "**cost of the bug**" has two meanings:

Meaning Number One: Expenses required to find the bug and fix it BEFORE the product's release – these expenses can be calculated.

Meaning Number Two: Damage to the business (plus the expenses required to fix the bug), because the bug was found AFTER the product's release – as a rule, it's extremely hard to calculate the exact monetary amount of the damage to the business

Meaning Number One:

If the bug was imbedded in the spec and the code was written with the bug, but the bug was discovered by the testers, the bug's cost can be* calculated like this:

Cost1: Hourly pay** of PM(s) **multiplied** by number of hours spent on buggy part of the spec

PLUS

Cost2: Hourly pay of the programmer(s) **multiplied** by number of hours spent coding buggy part of the spec

PLUS

Cost3: Hourly pay of tester(s) **multiplied** by number of hours spent for test preparations (e.g., test case creation) and testing of buggy code

PLUS

Cost4: Similarly calculated costs of spec fixing, bug fixing, test case creation/modification, and testing of the fix

PLUS

Cost5: Other possible expenses, e.g., expenses that might occur if release date slips

* This is a situation where the spec bug was due to wrong instructions in the spec. If the bug is due to an omission in the spec, then that bug's cost is calculated differently.

** The hourly pay of a consultant is usually specified in the contract; the hourly pay of a salaried employee is calculated as a total pretax salary per a given period of time divided by the number of hours during that period.

As you can see, it's possible to **approximately** calculate cost of a bug found before a product's release.

There are three important conclusions here:

1. **The earlier bug is found, the cheaper it is for the company.** This is the main conclusion.
2. The quality of the PM's work is critical because it takes place before coding and testing. In our example, if the spec had been bug-free, then Cost1 to Cost5 (inclusively) would not have occurred.
3. It's good to find bugs before the release, but it's much better and **cheaper** for the company if bugs are PREVENTED from happening in the first place. That's why we spend so much time talking about QA – i.e., about ways to prevent bugs.

Meaning Number Two:

If the spec bug wasn't found before the release, then the bug cost is calculated like this*:

Damage1: Cost1 to Cost5 (inclusively) from Meaning Number One

PLUS

Damage2: Time spent by customer support to calm down angry users

Damage3: Monetary compensation to users and business partners

Damage4: Loss of profits

Damage5: Law suits against the company

Damage6: Forever lost users and business partners

Damage7: Miscellaneous other negative consequences

* This is just an example. In **each case**, the damages can vary from fifteen minutes spent by company workers to fix, test, and release the bug fix to an extreme situation where the company's business is completely ruined.

As you can see, it's really hard to calculate an **exact** cost of a bug released to users.

Please understand that we are not playing games here. Here is a quote from Wikipedia: "A study conducted by National Institute of Standards and Technology (NIST) in 2002 reports that software bugs cost the U.S. economy \$59.5 billion annually. More than a third of this cost could be avoided if better software testing was performed."

The very important conclusion here is that **software companies MUST regard QA and testing as SELF-PRESERVATION mechanisms.**

Let's get back to unit testing. Here are two recommendations:

1. In the case of more or less complex/time consuming projects, *e.g., a project dealing with important architectural changes with an estimated coding time of 5 or more days*, **unit tests should be created BEFORE the code is written**. Then, when the programmer gets a new spec for coding, he or she doesn't rush in to start coding, but first creates the **code design document** and the **unit test document**.

Please note that "document" does NOT mean some multipage, highly formalized document ready to be printed in the Journal of Computer Science. After all, in the start-up phase, timing is everything, so both the code design and the unit test documents can be short memos. However, the programmer must make sure that those memos are:

- a. published on the Wiki;
- b. easily understood by others.

The beauty of this approach is this:

- **First**, when the programmer abstracts himself or herself from hands-on code writing, he or she has more intellectual resources to think about the "big picture". This way the programmer gives himself or herself a chance to predict – and hence *prevent* – mistakes in the future code.

- **Second**, the programmer will have to **imagine** HOW he or she will test his or her code, and that "HOW" will stick in his or her brain like a splinter. This way, the programmer will have a "quality guard" in his or her brain to **monitor** programming activities and thus prevent and catch bugs.

Brain Positioning

Often it's boring and laborious to create documentation, and the programmer is tempted to start programming right away without planning. But, IMHO, the main attribute of a real professional is doing what's right. So, if it's required by the situation, the company standard, or a verbal agreement to create documentation, that documentation **must be created**, and it **must be created properly**.

2. Requirements about unit testing must be formalized in Unit Test Standards. For example, there should be at least one test case for each Python function.

8. “IF IT AIN’T BROKE, DON’T FIX IT”

Every software engineer knows about cases where a perfectly working code becomes a nightmare after some “I-can-do-it-better” type of developer tries to improve it.

The problem that I'm referring to usually happens when a major chunk of the code is written and released; users are enjoying our product, and we can relax a bit from the craziness of the early start-up days. Some “I-can-do-it-better” type of developer (or any developer asked to do so by his or her manager) looks back at the old working code, finds room for improvement, and implements that improvement. And here come the problems!

Example

My friend John and I have just finished the testing of new functionalities and are going to spend two days testing old functionalities (this kind of testing is called **regression testing**). We've executed test cases for regression testing many times for previous releases and usually discover very few bugs. We start testing, and ten minutes later we simultaneously shout: "WHAT THE HECK IS GOING ON?" The reason for this behavior is that the state of our old functionalities reminds us of the state of a fine porcelain shop right after an elephant's visit – everything is broken!

After more investigation, we find out that there was a cute little project to rewrite several pieces of old code with the purpose of "purifying the breed." Well, better luck next time. This time, our application is screwed up, and thanks to the regression testing we have uncovered serious issues before releasing that ugly code to our users.

You might be surprised, but situations like the above happen in software companies over and over again. And all because some folks cannot comprehend programming wisdom as old as the world: "**IF IT AIN'T BROKE, DON'T FIX IT.**"

So why should programmers think twice before changing an old code just to beautify it? The reason is simple:

The old code is usually part of the software foundation in which many pieces of old code and newer code have been heavily integrated, thus creating a complex net of dependencies.

So, when we shake this foundation, as a rule we simply cannot predict which part of the application is going to be affected. That's why even a simple change in the old working code can have major negative consequences in totally unexpected parts of the software. This kind of effect is called "unintended consequences."

Unintended consequences during a code change are the major reason we need regression testing.

So, remember that it is usually a very BAD idea to change working code just to improve it. Simply put: "**IF IT AIN'T BROKE, DON'T FIX IT.**"

9. BEING LOVED BY THE COMPANY

It's really simple. Management should take care of people. Start-up employees are heroes, spending day and night at the office, working on their shared dream to create a super product and get rich.

Management should reward them for their hard labor and dedication. Is it too much for the company to give a \$100 gift certificate or an extra thousand stock options to the employee who finished some amazing project? Or is it really all that expensive to buy lunches for the employees at least once a week?

It's human nature to get discouraged when work is not appreciated. I'm amazed that so many managers don't understand this simple truth. Again and again, I hear stories about unfair financial treatment and the absence of financial encouragement. Who suffers because of it? **Everybody in particular, and the business as a whole!**

On the opposite end, companies where employees are treated fairly and encouraged to go above and beyond have MUCH more chances of creating a great product.

For a while I worked at the Plug and Play building in Sunnyvale, California. This is an incubator for new companies, with lot of software-related conferences taking place there. The tradition is that the speaker writes something on the wall on the first floor. What do you think one of Google's managers wrote? "It's all about the people."

Yep, without this kind of attitude Google would have been a smart search algorithm, nothing more. But because founders Sergey Brin and Larry Page built their company on the principle "***It's all about the people***", Google is the most successful company in the history of software development. Let me list some of the things available to Google employees at Google's Mountain View, California campus. **Most** of these things are FREE:

- 11 gourmet cafeterias,
- laundry,
- dry cleaning,
- oil change,
- car wash,
- gym,
- exercise classes,
- massage,
- Mandarin, Japanese, Spanish, and French classes,
- personal concierge to arrange dinner reservations,
- hairstylist,
- \$5,000 toward environmentally friendly car,
- \$500 in takeout food for new parents for first month at home,
- five on-site doctors,
- Wi-Fi-enabled shuttles from five Bay Area locations,
- breast pumps for nursing moms,
- heated toilet seats (!),
- requirement to spend 20% of work time for dream projects.

Source CNN Money, 01/05/2007

I forgot to mention that **ALL pre-IPO employees of Google became millionaires**, and several hundred thousand dollars is a possible bonus for a successful project.

I could go on and on about Google. Now you understand what I'm talking about when I praise Google as an employer that LOVES its employees!

*I realize that Google is a rich public company, and start-ups don't have such huge material resources. But think about this: each big company was once a start-up, and as we know, big companies very often suck as employers. So why is Google both a huge company **and** a great employer? Because both in Google's start-up years and now, it's always been **all about the people**.*

Here's what any start-up can do, even without Google's resources:

- pay bonuses for successful projects; even \$1000 is better than a moneyless "Amazing job, man!"

- provide free lunches
- provide free soda and bottled water
- provide free snacks
- reward with gift certificates or concert tickets
- give Christmas gifts like jackets with the company logo
- organize paintball parties, hiking, or rafting trips
- give promotions
- pay for memberships to a nearby gym

There are many options, and every dollar spent on employees can turn into hundreds of dollars, because people are basically good and **we DO work better in exchange for management taking better care of us.**

10. HAVING "QUALITY" AND "HAPPINESS OF USERS" AS FUNDAMENTAL PRINCIPLES OF COMPANY PHILOSOPHY

Management must make sure that everyone at the company understands that "quality" and "happiness of users" are not just abstract concepts but rather the way to success for the company, and hence a way to personal success for everyone who works for the company. If managers make fun of quality measures and make jokes about users (even during an informal chat!), it does real harm to the internal morale of the company, and in the end there will be negative consequences for the company as a whole and hence for those idiots-humorists in particular.

Users know whether or not they are respected after a single error message, email from the company, or call to customer support. And if the real philosophy inside the company is, "Our users are stupid," then that philosophy will sooner or later be evident to users. Of course, this will be great news for that company's competitors.

Our users are real people. They should not be perceived as a faceless crowd, but rather as friends who trust us and appreciate our hard work.

Let's proceed with "Coding" stage and talk about the two main things done by the programmer:

- 1. Programming.**
- 2. Bug fixing.**

Here is simplified version:

1. PROGRAMMING

A developer writes

```
child_value = parent_value + 3
```

2. BUG FIXING

According to the spec, the child value must be equal to the parent_value **minus 3**:

```
child_value = parent_value - 3
```

So, a tester files a bug about "+" instead of "-" and the programmer does the bug fixing.

Most of the PROGRAMMING takes place during the *Coding* stage. But PROGRAMMING can also take place anytime when there is a need in code creation/modification/deletion.

Most of BUG FIXING takes place during

- the *Coding* stage (developer finds his or her bugs) and
- the *Testing and bug fixes* stage (bugs are found by a tester).

But BUG FIXING can also take place anytime when new bug is found and deemed to be worth fixing.

BTW

What does a developer do when the programming for the coming release (e.g. 1.0) is finished? He or she writes the code for the next release after coming release (2.0)! So, when testers find bugs during stage "Testing and bug fixes" developers have to interrupt their work on 2.0 and fix bugs for 1.0.

*The important thing to understand here is that after the developer has done his or her work for the current Cycle, he or she moves to the "*Coding*" stage of the next Cycle.*

Now, let's learn about three classic* types of bugs associated with coding:

1. Syntax bugs
2. User Interface (UI) bugs
3. Logical bugs

* We are not going to cover highly specialized and advanced topics like performance testing.

1. SYNTAX BUGS

The beauty of syntax bugs is that they are detected and even explained (sort of) by a language compiler (e.g., if the programming language is C++) or a language interpreter (e.g., if the programming language is Python) once you try to compile or execute the program.

BTW

Compilers and **interpreters** are special programs that translate code typed by humans ($a=2+3$) into the language understood by computers (011000010011101001100100010101100110011). The difference between a compiler and an interpreter is that:

- a compiler produces an executable file but doesn't execute our code. Thus, text files written in C++ must be compiled to become executable (launchable) programs.

- an interpreter doesn't produce an executable file but immediately executes our code. Thus, text files written in Python are already executable.

Example

Let's look at the example of a syntax bug in C++ program. C++ uses compiler.

Here is first program of any student learning C++ (line numbers are not part of software code):

```
1. #include <iostream.h>
2.
3. void main()
4. {
5.     cout << "Hello, World! << endl;
6. }
```

This code is saved in the file hello_world.cpp. Let's try to compile this program:

```
~> gcc hello_world.cpp
hello_world.cpp:5: unterminated string or character constant
hello_world.cpp:5: possible real start of unterminated constant
```

The last two lines are the compiler's description of the bug in this code. What happened was that the compiler found a problem and exited after giving us information about the bug. The bug happened because we didn't close with double quotes after *World!*. How would we describe this bug in one sentence? "**hello_world.cpp: closing double quote is missing after World!"**"

If we fix the problem, our hello_world.cpp will compile just fine, and we will get an executable file that prints "Hello, World!" when we launch it.

Example

Let's look at the example of a syntax bug in Python program. Python uses interpreter.

SL Go to this URL: http://www.sharelane.com/cgi-bin/register_with_error.py (you can also launch this file from here: Test Portal>More Stuff>Python Errors>register_with_error.py). As you can see, there is a syntax error in this line of code:

```
body_html = get_firstpage(zip_code)
```

Programmer made mistake by giving wrong name of function:

Expected: get_first_page
Actual: get_firstpage

Here is a bug summary: "**register_with_error.py: wrong name of the function get_first_page()**"

Brain positioning

SL The only difference between

- perfectly working file register.py (Test Portal>Application>Source Code>register.py) and
- not working at all file register_with_error.py (Test Portal>Application>Source Code>register_with_error.py)

is a single underscore character ("_"). So, even smallest modification to existing code can break the whole system. That's why regression testing, i.e., re-testing of old functionalities, has immense importance.

2. USER INTERFACE (UI) BUGS

A UI bug is a bug in **how the software presents the information**. UI bugs range

from **visual problems** like

- *a link on the Web page has a wrong color*

to **interactive problems** like

- *it's hard for a user to figure out how to use certain functions.*

So remember that UI bugs cause **presentation** problems.

3. LOGICAL BUGS

A logical bug is a bug in **how the software processes information**.

Example

SL Do steps below:

1. Create new user account on main.sharelane.com and login.
2. Add any book to the Shopping cart.
3. Set number of book inside Shopping cart to 30 and click button "Update".
4. Check value in column "total".

We have logical bug inside functionality "Shopping cart", because total amount got **calculated** incorrectly:

Expected result: 294 (discount must be **subtracted**).

Actual result: 306 (discount is **added**).

BTW

SL Let's see buggy line of code. Go to Test Portal>Application>Source Code>shopping_cart.py and click link "BUG #6" under sub-section "View bugs."

Expected statement: total = total - discount_amount

Actual statement: total = total + discount_amount

Logical bugs are the primary focus of software testers because

- As a rule, it's much harder to find logical bugs than UI bugs;
- As a rule, the consequences of releasing logical bugs are much more severe than the consequences of releasing UI bugs.

But keep in mind that even UI bugs can be really nasty: if users cannot figure out how to put a book into the shopping cart, then the perfectly working code of the application core doesn't really matter.

Here is another stage-by-stage illustration of a logical bug:

Example

<Product design>

Spec #9877: "7.2. User must enter two integers from 1 to 12 and press enter. Program must print on screen arithmetical mean between entered values."

<Coding>

Here is the code written in C++. That code, also called a **source code**, is in text file get_average.cpp.

```
01. #include <iostream.h>
02.
03. void main()
04. {
05.     int first_number = 0;
06.     int second_number = 0;
07.     float average = 0.0;
08.
09.     // get first number
10.     cout << "Enter first number: ";
11.     cin >> first_number;
12.
13.     // get second number
14.     cout << "Enter second number: ";
15.     cin >> second_number;
16.
17.     //calculate average
```

```

18.     average = first_number+second_number/2.0;
19.
20.     //output result
21.     cout << "Average = " << average << endl;
22.
23. }
```

Now, after the code is written, the programmer uses the C++ compiler **gcc** to compile the text file `get_average.cpp` into executable file `get_average`.

<Testing and bug fixes>

```

~>get_average
Enter first number: 9
Enter second number: 2
Average = 10
```

According to the spec, our expected result is 5.5 if we use 9 and 2 as inputs. But our actual result is 10, so we have a logical bug. What will you do now?

- You will file a bug against the responsible programmer. Bug summary: “**Spec9877: get_average: wrong calculation of arithmetical mean**”.
- Then the programmer should look into the problem, fix it, and **reassign** the bug to you (soon we’ll cover all the technicalities related to bug tracking).
- Then you must retest the code and
 - if the bug is fixed, you close it;
 - if the bug is NOT fixed, you reassign it back to the developer.

BTW

The problem with the code was in line 18 where our programmer didn’t put a parenthesis. Here is the correct expression:

```
average = (first_number+second_number)/2.0.
```

BTW

We should file another bug against the PM who didn’t specify the precise range of inputs. If the max valid input is 12, than the first sentence of item 7.2 must have this wording: “User must enter two integers from 1 to 12 **inclusively** and press enter.” Why do we need “**inclusively**”? Because without it:

- Some folks will be confident that “integers 1 to 12” means that the max value is 12.
- Some folks will be confident that “integers 1 to 12” means that the max value is 11.
- Some folks will be plainly confused.

So, while writing specs the PM must make sure that everyone in the spec audience will understand the spec the same way.

BTW

That spec has another problem: it's not specified how the program should react to invalid input, e.g., 0, 13, "A", "#" or Null input (the user simply presses Enter without any data typed).

Let's proceed and talk about **code freeze**. Why do we need it? **It only makes sense to test code if the code is in a stable state – i.e., the developers don't change it while it's being tested.**

Example

Imagine the following situation:

1. The programmer has finished coding functionality A.
2. The tester checks functionality A and gives green light for release.
3. Ten minutes before the release, the programmer remembers something and makes a tiny change in the code of functionality A.

Theoretically, that "tiny change" should NOT affect anything, BUT **in reality**, functionality A stops working completely.

So we have a situation where

- the tester just wasted his or her time testing a not yet final version of the code
- the users got a buggy Web site

Remember this for the rest of your testing career:

Before you start testing, make sure that

- 1. The code is frozen (release engineers usually send an email stating this).**
- 2. The test environment* has the correct version of the code to test.**

**The test environment is the software/hardware combo where the software is tested before releasing it to users. Usually, when referring to the test environment we just mention the name of an internal Web site for testing. If the URL of the test environment is <http://main.sharelane.com>, then I can say: "I found a bug on main."*

Remember to check these two things EVERY time prior to testing.

Let's illustrate the second point.

Example

We have two test environments on the Intranet:

<http://everest.sharelane.com>
<http://elbrus.sharelane.com>

These two Web sites are used for internal testing only and are not available to users.

Let's assume that

<http://everest.sharelane.com> has code version 1.0.
<http://elbrus.sharelane.com> has code version 2.0.

There is often a situation when the same functionality (let's say "Checkout") in the old (1.0) and new (2.0) versions has

- an absolutely identical front end and
- an absolutely different back end

Therefore, if a tester doesn't **specifically** check to see if the test environment has the correct version, he or she can end up testing the wrong version, because the front end looks the same. This mistake is often made by both beginning and experienced testers. That's why it's very important to *always* check to see if the test environment has the correct version, i.e., the exact version that needs to be tested.

So what happens if a tester needs to test version 2.0, but erroneously uses <http://everest.sharelane.com> for testing? Two things:

1. **A waste of that tester's time** – if we are going to release version 2.0, but erroneously test version 1.0, it doesn't make much sense.
2. **The risk of releasing untested code to our users**, because version 2.0 (which was supposed to be tested) wasn't tested.

As you can see, **negligence in checking the version of the code to be tested can have very negative consequences**.

My advice to you is this: If you screw up and test version 1.0 instead of version 2.0, immediately acknowledge your fault. Everyone makes mistakes. It's life. Just forget about your plans for the evening, weekend, or holiday, and stay in the office until you finish testing version 2.0.

BTW

It's a good idea for a start-up employee to have a sleeping bag, a change of clothes, a towel, and some personal hygiene items ready in the office – there is always the chance for some wild overtime, and you don't want to drive after fifteen hours of work.

Sad things, like testing of wrong version, take place because of a tester's negligence, but sometimes other factors, like illogical names of test environments, contribute to the situation. Here's how to prevent this:

1. Ask your release engineer how to find out the current code version in the test environment and apply this knowledge **EVERY** time before testing.
2. Ask your IT engineer about choosing logical names for test environments. An IT person can enjoy whatever he or she likes (for instance, mountain climbing), but it's difficult to understand the difference

between Everest and Elbrus when talking about a version of software. Let's create two test environments instead:

http://old.sharelane.com for the application version which is currently on our live site: <http://www.sharelane.com>. So, if a bug is released to our users, we have <http://old.sharelane.com> to test a bug fix before releasing that bug fix to the live site. *Between ourselves we can call this test environment "Old".*

http://main.sharelane.com for the application version of the coming release; the testers do testing on <http://main.sharelane.com>, and when we are ready for the release, the live site, <http://www.sharelane.com>, will get the version of the software accepted for release on <http://main.sharelane.com>. *Between ourselves we can call this test environment "Main".*

3. If the current way of finding out the version of the code and the DB is not convenient (e.g., you always have to go and ask the release engineer about those versions), suggest that he or she implements one of the following:

- a. Add a Web page with information about
 - the release version
 - the build number (more about this in a minute)
 - the DB schema version (more about this in a minute);

SL For example, if this is implemented, you can go to the page: <http://main.sharelane.com/version.txt> and see something like this:

Release version:	1.0
Build number:	23
DB schema version:	34

Important: the "version of the code" or the "version of test environment" consists of the three components above. Remember to check for these three things every time before you start testing. We'll elaborate more about each of the components soon.

b. Another good approach is to include information about the current version as a comment written into the html code of each Web page.

Example

SL Do this:

1. Go to main.sharelane.com
2. Point your cursor on an object-free spot of the Web page; an "object-free spot" is any spot on the Web page without a Web object (such as an image, link, etc.).
3. Do a single right click.
4. Select the "View Source" (IE) or "View Page Source" (Firefox) option
5. Find the version number.

For example:

```
<!-- application version 1.0-23/34 -- >
```

You can find out application version of ShareLane looking into HTML of **any** Web page on <http://main.sharelane.com> (exception is Test Portal pages).

BTW

From now on, let's call the full version of our code the "**application version**".

At ShareLane, we specify application version in this format:

```
<major release number*>.<minor release number>-<build number>/<DB version number>
```

* we'll talk about release types later.

Three last things about "Coding" stage:

First: During the "Coding" stage, programmers write code for the coming release and testers write test cases to test that code. I recommend that files with test suites, just like files with specs, should be

- stored in CVS (or other version control system);
- available to anyone inside the company.

The main advantages of storing files with test suites in CVS (or other version control system) are:

- it eliminates the risk of an accidental deletion of a file;
- it allows access to old editions of the file and its history;
- the file is stored on the shared server, and everyone who needs to (and who has a right to do so) can take that file to
 - > execute, modify, or delete existing test cases;
 - > append new test cases.

Second: It's very good idea to have a tradition of test case review meetings. Test case review meetings take place before the stage "Testing and bug fixes" and right after the tester has finished writing test cases. What happens is that the following people:

- the PM who wrote the spec,
- the programmer who is responsible for coding that spec and
- the tester who has finished test cases to test that code

get together, and the tester makes a mini presentation about the way he or she is going to do testing.

Before the meeting I usually make hard copy of the test suite for everyone who accepted my meeting invitation. During the meeting I give a short overview of each test case, and everyone can see what I'm talking about.

The great value of this kind of meeting is that in many cases PMs and programmers

- give testers new IDEAs for testing *and/or*
- find omissions and/or errors in test cases.

BTW

After the test case review meeting, send out an email to every participant and to those who were invited but couldn't come to the meeting. In this email, list all improvements that you've talked about during your meeting. This email is very useful because

- you'll refresh the suggested ideas in your brain;
- you'll give others an opportunity to see that you understood them correctly.

In many large Internet companies, the test case review meeting is an obligatory thing.

Third: After my wife read about UI and logical bugs, she asked me a simple question: **If an HTML link leads to the wrong page, is it a UI or a logical bug?**

Interesting question:

- on the one hand, it's about presentation;
- on the other hand, sometimes links are dynamically generated by our code (e.g., the link "log out" is generated only if user is logged in).

So, my response was:

- "**If the misleading link is *hard-coded* into the HTML code of the Web page, then it's a UI bug**, because there is no problem with processing here and the presentation is wrong."
- "**If the misleading link is *dynamically generated* by our code, then it's both a logical and a UI bug**, because there is a definite issue with processing and there is definite issue with presentation."

Does this make sense? What do you think?

That's all for the "Coding" stage, my friends. I hope that you've learned some valuable stuff that you will be able to use in your companies.

Testing And Bug Fixes

We are going to cover a lot about this stage during the rest of our Course. So, for now, let's be as laconic as Spartans.

After the developers have finished writing the code, the release engineers make it available for the testers in the test environment (<http://main.sharelane.com>). The testers perform a quick **smoke test** (also called a **sanity test** or **confidence test**) to check if the software is **testable**.

Example

If we cannot log into the account on main.sharelane.com, we simply cannot proceed with test execution, i.e., **the software is not testable**.

As a rule, a smoke test doesn't take more than half an hour, and it usually doesn't require any special test cases.

If the smoke test fails, we communicate this to the programmers and release engineers (there can be an error on the release engineering side), and they work to fix the problem. Once the problem is fixed, we perform the smoke test again. This mini cycle takes place until the smoke test passes. Once the smoke test passes, the release engineers freeze the code and push it to the test environment (main.sharelane.com), and the testers start the **new feature testing** by executing test cases written for this release.

Brain positioning

“Feature” is a much wider term than “functionality” which we’ve been using so far. **Functionality is the ability to accomplish some task.** *For example, the functionality (ability) of a bottle opener is to open bottles.*

Depending on the context, the term "feature" means

- the ability to accomplish some task or
- some characteristic of the software.

For example,

- the Shopping Cart can be called both a "feature" and "functionality", because it is an ability that allows users to store items they want to buy.
- the color of the Shopping Cart link can be called only a "feature", because the color of that link
 - >is a characteristic of the software and
 - >it does not have functional (i.e., task-solving) aspects.

When in doubt, use the term “feature.”

BTW

A favorite developer's expression, "It's not a bug, it's a feature", in human language sounds like, "That something is not a problem with my code. It works (and/or looks) exactly as I want."

After the new feature testing is finished, the testers start to test the old features. This type of testing is called **regression testing**. Regression testing is performed to check if code modifications made for this release have broken something in the old features. After we are finished with new feature testing for our release 2.0, we are going to use our test cases written for release 1.0 for regression testing.

BTW

Does this mean that during regression testing for release 25.0, we have to execute all test cases from releases 1.0 to 24.0 inclusively?

Well, this is one of the most difficult questions in testing, and we'll address it later.

Found bugs are filed into the bug tracking system, the programmers fix them, and the testers verify those fixes.

Once the regression testing is finished (there is usually a concrete deadline), testers perform an **acceptance testing** (also called a **certification testing**). During acceptance testing, the testers:

- execute a special set of P1 test cases (often in form of checklists)
- do ad hoc testing (undocumented testing based on intuition and inspiration)

Sometimes, when a young start-up has an important release (e.g., its first release), the whole company stays overnight to do acceptance testing.

Once the acceptance testing is finished, a gang of brothers and sisters (PMs, testers, developers, release engineers, managers) get together for a Go/No-Go meeting. At this Go/No-Go meeting the group decides if the code is ready to be released to the production environment (also called "**production**," "**prod**," or "**live**") so the users can enjoy the results of the team's efforts. If it's a "Go," the release engineers release the code to prod. If it's a "No-Go," the responsible parties stop eating and sleeping until the issues are resolved. Once all the issues are resolved, there is another Go/No-Go meeting, and so on ... until release to prod takes place.

Release

Example

In the spirit of Steven King's novel, a little boy – a dreamer, a book lover, and an insect collector – is being constantly humiliated by his siblings, classmates, and accidental bystanders. One day he says, "Enough is enough," and starts to cut, shoot, strangle, and burn his abusers, and for the sake of prevention, all others who are in his way. This situation is about letting the steam out – or "release" in everyday terms.

Luckily for us, in the software industry the term "release" is used in a totally different way:

- as a **verb**, "**to release**" means **to transfer a piece of software to the users**. For example, we can ask a release engineer to release the code to production.
- as a **noun**, "**release**" means **a certain piece of software**. For example, we can say: "We are testing release 5.0."

BTW

We'll apply the term "**production version**" or "**version of prod**" to the release that is in production now.

We'll apply terms "**coming release**" to whatever version we are going to release next.

The important thing to understand is that **a release is not some kind of abstract software; it is a package of concrete files having concrete versions**. Please pay attention.

Example

The release 5.0 contains 63 files. Each of these 63 files has its own version. When the testers have finished testing release 5.0, the release engineer will release precisely **the same 63 files** that were in the testing environment when the testers finished testing. If the testers finish the acceptance test on January 03 at 4:00 pm, and at that time the file register.py had its own version 5.34, then it must be exactly version 5.34 of register.py that is included into the release 5.0 when 5.0 goes to production.

BTW, the version number is automatically assigned to file by CVS (or whatever version control system is used) every time a developer saves (or more professionally, “commits”) updated version of the file to CVS.

The purpose of the release is to transfer one or a combination of the following things, to production:

- 1. New features.**
- 2. Modification/removal of existing features.**
- 3. Bug fixes.**

There are two main types of releases:

1. A major (or milestone) release happens at the “Release” stage of the Cycle, after the “Testing and bug fixes” stage is over; i.e., “Go” decision was made at “Go/No-Go” meeting.

Besides files with codes containing the application core, html files, images, etc., major releases also contain SQL scripts to be uploaded and run on production DB servers. For example, if a developer wrote a code that uses a new table in the DB, he or she needs also to create an SQL script that has to run in the DB to create that table.

The version of a major release is presented as an integer: 7.0

2. A minor release takes place between major releases. Minor releases can have one of three variants:

- **feature release;**
- **patch release;**
- **mixed release.**

A FEATURE RELEASE takes place when there is a need to

- add new features;
- modify/remove existing features.

A PATCH RELEASE takes place when the code in production has a bug (or bugs). Here we release a fixed code.

BTW

Please note that discovering a bug in production doesn't mean that the users have already run into it.

For example, let's assume that minor release 7.1 took place at 2:00 a.m. This release contained a special tool for bank payment processing called `process_payments`. The tool is scheduled to run at 1:00 am every day. So, there is at least 23 hour window during which we can find and fix a bug in `process_payments` before our users suffer from it.

BTW

In some cases, a bug is found "theoretically": e.g., the PM or developer suddenly wakes up in the middle of the night and asks himself or herself, "What if we have this situation: <...>? Can our code handle this?"

BTW

There are two cases when we have a bug in production:

a. **The bug has non-P1 priority**; e.g., the code of `register.py` doesn't check if the email entered by the user has a dot (".") before the top-level domain. In this case, there is no emergency to release a bug fix ASAP. So:

- we can "accumulate" several non-P1 bugs and do a single patch release with bug fixes for all of them OR
- we'll just wait until the coming major release (where all those bug fixes are present) is pushed to production.

b. **The bug has P1 priority**; e.g., the user cannot register at all. In this case, we have **an emergency**. A patch release is initiated, created, and released by the rules set in the **Emergency Bug Fix procedure (EBF procedure)**. The patch release for the **Emergency Feature Request (EFR)** must also follow EBF procedure. You'll read more about EBF and EFR in a minute. *See example of EBF procedure under Downloads section on qatutor.com.*

A MIXED RELEASE is a minor release that occurs when there are both feature related changes and bug fixes.

The version of a minor release is presented as a number after a decimal point and incremented by one after each minor release: 7.1

Here are several differences between major and minor releases:

1. **The main difference is that, as a rule, major releases have tons of new features/bug fixes**, while minor releases usually contain only one new feature/bug fix. *For example, after the release of version 5.0, we signed a contract with the credit card processor so we can accept another credit card - Discover. In that case we can do a special feature release 5.1 to add just one functionality ("users can pay with Discover") to our Web site.*

2. As a rule, major releases have a recurring schedule; e.g., one major release per month or per quarter. Minor releases happen whenever they are required.

3. Major releases exist in the main Cycle, while minor releases exist outside of the main Cycle.

Brain Positioning

So far we have used the term “Cycle” as a companywide activity, the purpose of which is to create a major release. But “Cycle” can also be applied to software development activities on a much smaller scale. *For example:*

- At 10:00 a.m. developer Josh was writing code for 7.0. and tester Chad was testing code for 6.0.
- At 11:00 a.m. developer Andrew discovered a bug in the production version, 5.0.
- At 11:15 a.m. Josh and Chad were assigned to work on a patch release.

At 11:15 Josh and Chad started working within their own mini Cycle for patch release 5.1., which exists outside of the Cycles for major releases 6.0 and 7.0.

4. A major release cannot be considered as a maintenance release for the production version, while some minor releases, e.g., patch releases, have purely a maintenance nature.

5. A major release is always a planned event, while a minor release can be both **planned** and **unplanned**.

A **planned minor release** usually happens:

- if there is a certain feature that couldn't be included in the major release due to resource/time constraints

For example, we really loved the features from the spec #1478 “Improvements for Shopping Cart,” but we didn’t have time to develop and test it for release 5.0 which had to be pushed to production on March 15th. So, we just release 5.0 and begin work on 5.1, which will contain features from 1478.

- if code of major release had known non-P1 bugs prior to that release, and we agreed to fix them in a patch release after the major release is out.

- if we've discovered several non-P1 production bugs and decide it's time to do a patch release.

An **unplanned minor release** usually takes place if there is:

- an emergency bug fix;
- an emergency feature request.

BTW

An **Emergency Bug Fix (EBF)** is a situation where a P1 bug is found in production and we need to push a patch release ASAP.

An **Emergency Feature Request (EFR)** is a situation where we need to release a certain feature ASAP; e.g., in case of a court ruling or to comply with a new law. *For example, our competitor won a*

patent case, and so we have to change some piece of our software ASAP to make it work in a different way.

Minor releases for an EBF and an EFR are treated as patch releases.

In case of an **EBF**, an entry with the type “**Bug**” and the priority P1 is entered into the bug tracking system.

In case of an **EFR**, an entry with the type “**Feature request**” and the priority P1 is entered into the bug tracking system.

BTW

In order to address possible EBFs after a major release, many companies create SWAT* teams which consist of a developer, a tester, and a release engineer. Each team has a twenty-four hour period when each member of the team must be available to come to the office at any time. Each team member whose team is on duty must have his or her cell phone on at all times and must refrain from doing stuff like drinking too much tequila or going skiing at Lake Tahoe. When a call about an EBF is received by a SWAT team member, he or she must drop whatever he or she is doing, come to the office, and do his or her job until the patch release is out, or until the next SWAT team arrives.

* *In the real world, a SWAT (Special Weapons And Tactics) team is a specially equipped, quick response police unit.*

Here are a couple more things about version numberings. What does production version 9.14 tell us? Basically, two things:

1. We had 9 major releases.
2. We had 14 minor releases AFTER the 9th major release was pushed to production.

What is the version of the next minor release, if any? 9.15.

What is the version of the coming major release? 10.0.

BTW

Please note that this format

<number of major releases>.<number of minor releases>

is the most common way to track release versions, but each company can use whatever format they like.

BTW

At some companies, there is a tradition of giving major releases *names* instead of *numbers*. There are basically two reasons for this:

1. Start-up folks have creative minds, and it's kind of cool to call a release “Sunrise” instead of “1.0.”

2. When people work on something with a meaningful name, they develop a personal attachment to the project. Naturally, it's more inspiring to work on "Muse" than on "5.0."

While I totally agree with these two points, my recommendation is simple: DON'T DO IT.

For example, company N. traditionally names their major releases using pop groups/singers. Below is a dialog between two friends, Anthony who works for start-up company N. and his friend Steve who just got a bottle of Hennessy Paradis and box of Padilla Habano cigars:

- *Wassap, Tony? It's Friday night. Are you coming or what?*
- *Nah, Stevo. I'm hanging with "Jessica Simpson."*
- *WHAT?!*

In my opinion, the only benefit for company N. to give the name "Jessica Simpson" to their major release is the enormous respect that company employees get from their pals outside the company. But apart from that, it's a bad idea.

The first reason is that it's not clear which release was first – "Paul McCartney" or "John Lennon." But it's crystal clear that "4.0" came after "3.0."

The second reason is that it sounds like crap when you say things like: "We are going to release a patch to Louis Armstrong." Come on, Louis Armstrong is a legend, and it's almost blasphemy to associate his name with a trivial patch release.

The third reason is that your business partners will understand what "5.0" means, but they would be in a little frustrated if you tell them: "We are going to implement a new payment functionality in 'Animals.'"

BTW

In case of very aggressive release schedule (e.g. weekly or bi-weekly), a very good approach is to call any coming release "Release 1" (or just "R1"), with subsequent release called "Release 2" (or "R2"), and so on. Production release is referred to as "R0".

So, if we release every Wednesday and today is Friday, R1 is release that we test now and that is to be released to production on Wednesday next week.

This principle is very useful because:

1. There is often confusion when "next" is used to refer to a release, because some people understand "next" to mean "coming release," while others think it means "the release following the coming release."

Similar situation was perfectly illustrated in the Seinfeld's episode "The Alternate Side":

Sid: Well I'm going down to visit my sister in Virginia next Wednesday, for a week, so I can't park it.

Jerry: This Wednesday?

Sid: No, next Wednesday, week after this Wednesday.

Jerry: But the Wednesday two days from now is the next Wednesday.

Sid: If I meant this Wednesday, I would have said this Wednesday. It's the week after this Wednesday.

2. In the case of frequent releases, it's natural to refer to a particular release in a related manner - e.g., R1 rather than its formal ID, 54.0.

Let's proceed.

Release infrastructure (CVS, etc.) and the actual push of released software to production is the responsibility of release engineers (REs).

Now it's time to dive into the more technical stuff, and I'll try to make it easy and fun for you. Just stay focused.

Let's imagine that our company, ShareLane Super Duper, Inc., was created to sell books via the Internet, and it has just received its first round of financing. Not much: only 5 million of rapidly depreciating U.S. dollars.

In the beginning, we have:

- two programmers: **Billy** and **Willy**;
- CEO **Jean Batiste Emmanuel Zorg** (further referred to as "Mr. Zorg" or "Evil Boss");
- two desktop computers with Windows OS for programmers (for the sake of simplicity, I will not provide versions of third-party software);
- the ultra slim, cool-looking laptop of Mr. Zorg (OS doesn't matter);
- one server (known as "the Star") with Linux OS for the development/test environment

In fact, Billy wanted to name this server after his cat: "Borborygmus," but after Willy and Mr. Zorg asked him a sobering question: "Are you crazy?" it was decided that they would have to be practical and give their servers beautiful, but completely understandable names, like "the Star."

BTW

On the Star, we have five test/dev environments:

<http://old.sharelane.com> - here we have the same version that's on the production machine. If prod has version 1.0, this environment has version 1.0.

<http://main.sharelane.com> - here we have a version of the coming major release (e.g. 2.0) if code for that release has already been frozen for testing. Otherwise, prod, Old and Main will have the same version (e.g., 1.0)*

<http://dev.sharelane.com> - here is where programmers do integration between their code before that code is delivered to main.sharelane.com. Any kind of version can be here.

<http://billy.sharelane.com> - Billy's playground. Any kind of version can be here.

<http://willy.sharelane.com> - Willy's playground. Any kind of version can be here.

* In fact, we've just released our version 1.0, so www.sharelane.com (prod), old.sharelane.com and main.sharelane.com will have the same version.

SL

See complete list of ShareLane environments here: Test Portal>Release Engineering>Environments.

The project STARTS!

1. We register the domain name sharelane.com.
2. We rent the server for the production environment at the hosting provider.
3. We unite all our local computers (Billy's machine, Willy's machine, the Star, and Zorg's laptop) into our Intranet.
4. The programmers start working on the code.

As you already learned, classic Web project architecture has these three components:

- **Web server**
- **Application core**
- **DB**

BTW

Because we've just started, all our test/dev environments will reside on the Star. Please note that each of those 5 environments will have **its own** Web server, application core and DB.

Now let's talk about sharelane.com architecture in detail.

1. APACHE WEB SERVER

The name "Apache" comes from "a patchy" because of the enormous number of patch releases applied to this free software. However, those patch releases did good, because Apache is a very reliable, high-quality software package. In the Apache directories we store

- HTML and JavaScript files: JavaScript code (or reference to the file with JavaScript) is incorporated into the HTML code, and it serves many purposes, from enhancing user interface to checking Web forms. On ShareLane we have JavaScript file timer.js called during user login (see Test Portal>Application>Source Code>log_in.py).

BTW

The advantage of using JavaScript to check Web forms is that this check (for example, for a valid format of email; e.g., no "@@") happens on the user's computer rather than on the production server. This way, we can reduce a load on the production environment.

- Images: For example, .GIF and .JPEG files; e.g., file logo.jpg:
<http://www.sharelane.com/images/logo.jpg>.

2. APPLICATION CORE IS WRITTEN IN PYTHON

Python scripts reside in a special directory in Apache called "cgi-bin." You can look into actual software code of the application core here: Test Portal>Application>Source Code.

3. DATABASE MYSQL

In DB we'll store data about users, books, orders and other things. See all current data inside ShareLane DB here: Test Portal>DB>Data.

Brain Positioning



Please note that we have to distinguish DB schema from DB data. The DB itself is a set of virtual containers called tables (actually, there is MUCH more to the DB, but for now we just need the basics).

Let's look at the table "cc_transactions" (Test Portal>DB>Data>cc_transactions), where we store data about the success/failure of each credit card transaction, i.e. each attempt to use credit card for purchase.

The cc_transactions table has 4 **columns**: id, result, user_id and order_id.

After each new transaction when the user attempts to buy a book, a new row is inserted into cc_transactions. These rows are called "DB rows", "DB records", or simply "rows" or "records".

The value of the column **id** is populated automatically with each new credit card transaction.

The value of the column **result** consists of two concatenated (joined) values:

<internal id of credit card, e.g. "1" for Visa>

and

<success code of transaction: "0" for success and "1" for failure>

; e.g., '10' means that transaction with Visa* was successful.

*you can see all card ids here: Test Portal>DB>Data>cc_types.

The values for columns **user_id** and **order_id*** are equal to the corresponding ids from tables users (Test Portal>DB>Data>users) and orders (Test Portal>DB>Data>orders); i.e. if purchase was made by user whose id in table users equals "1220", value of user_id in corresponding record of table cc_transactions will also be equal "1220".

* in case of failed credit card transaction, order_id value is equal "0" (zero), because no order has been made.

Now, let's simulate a book order where a user enters his or her MasterCard and the payment was successful.

1. Create new user account on main.sharelane.com.
2. Buy any book using MasterCard and write down Order id.
3. Go to Test Portal>DB>Data>cc_transactions and search web page for your Order id (you should find value of your Order id under column order_id).

4. Get value of column "result" from the same DB record.

Expected result: 20

As you can see, actual result is "10", so we got a bug!!!

What is the bug summary? How about this: "**Checkout: wrong value in result column of cc_transactions when using MasterCard**".

Let's do another cool thing now. Let's file that bug into our bug tracking system: go to Test Portal>Bug Tracking>Training BTS>Submit New Bug. Fill up only Summary and Description (put steps to reproduce the bug) before submitting the bug.

CONGRATULATIONS, you've just filed your first bug!!!

Let's proceed.

The DB schema is about containers: tables, columns, etc. The DB data is about the content of these containers.

So,

- if we add a new column time_created to the table cc_transactions or create a new table, we'll change the DB schema.

- If we do any manipulation with the content of the table cc_transactions, e.g., create a new transaction that would insert a new record into cc_transactions, we'll change the DB data.

The best analogy is this: **The DB schema is a plastic bag; the DB data is water in this plastic bag.**

The DB schema has its own versioning, usually starting with 1 and incremented by 1 with every schema modification. So, if we have a version of the DB equal to 34, then after we modify the DB schema (e.g., add a new column to the cc_transactions) we'll have DB version 35.

The DB schema is created/modified

- manually – e.g., developer runs SQL statement (-s);
- by creating and running **SQL procedures** which incorporate those SQL statements.

SQL statements/procedures that modify DB schema must be checked into the CVS.

People who work professionally with databases are called DB Administrators or DBAs for short.

You can see DB Schema version 34 of ShareLane.com here: Test Portal>DB>Schema.

As our software development begins, the first problems emerge because neither Billy nor Willy has an experience in version control. For example, they can simply delete each other's work if they save a file with the same name in the same directory, e.g. if Billy copies his version of register.py into directory

/var/www/dev/cgi-bin and then Willy copies his version of register.py into the same directory, Billy's version is overwritten and gone from /var/www/dev/cgi-bin.

So, Billy, Willy, and Evil Boss make an historic decision to use CVS.

BTW, the great news is that CVS is FREE, and it came preinstalled on the Star. Here are some details about this CVS:

1. Versions of each file are stored in CVS repository.

We can retrieve the needed version of the file from the CVS repository to view/edit it (this operation is called “checkout”).

We can place a new version inside the CVS repository after creation/editing of the file (this operation is called “checkin”).

Here is how we do this:

- addition of **new file to CVS**: `cvs add register.py`
- **checkout** of the latest version of register.py: `cvs co register.py`
- **checkin** of the new version of register.py: `cvs ci register.py`

2. When we checkin a new version of the file (the file is considered to have a new version even if we added/deleted a single character, like "#"):

- We don't need to change the file name.
- CVS automatically assigns a unique version number to that particular version of the file.
- During checkin, CVS allows us to make comments for this specific version of the file.

- During checkin, CVS also stores the version number, comments, and name of the person who did the checkin and the time of checkin. So, not only we can see all the checked in versions of the file, but we also can see all of that information for each version. How cool is that!

Now, after we have all of the application files checked in into CVS, we have another task: we need to have the files from CVS available in our test environment, <http://main.sharelane.com>. In order to do this, we need 2-steps process:

Step 1. Checkout from CVS **all application files for a specific release**. In other words, we need to get an **image (reflection) of the latest CVS content for a specific release**. This image is called a **build**.

Step 2. Transfer those files to the corresponding directories in a certain environment (e.g., if we want to have files for the coming release in our test environment, main.sharelane.com, we must use these directories:

- `/var/www/main/htdocs` for HTML and JS files;
- `/var/www/main/htdocs/images` for images;
- `/var/www/main/cgi-bin` for application core (Python files).

Let's elaborate on Step 1. Here is the definition for the term “build”: **Build is a sub-version of the specific release**.

Example

Let's say that our coming release is 4.0 and our application consists of only two files: **index.html** and **register.py**. In CVS we have version **4.11 of index.html** and **4.23 of register.py**. So, the package that consists of version **4.11 of index.html** and **4.23 of register.py** is a **build**. What if Billy changes **register.py** and checks it into CVS, so we have version 4.24 of **register.py**? In that case, the CVS image for 4.0 will be different, and thus, next build (i.e., sub version of 4.0) will be different from previous one.

Builds are created and pushed to a target environment, e.g., to main.sharelane.com, by a special build script, written by the RE to automate their job.

After a code freeze, a build script is often added to cron (the task scheduler on a Linux system) to create and push builds in equal intervals of time, e.g., every three hours.

SL See Test Portal>Release Engineering>Build Schedule.

The purpose of creating new builds over and over again is to make a modified application available for testers.

Example

Let's say that you've found a bug during your testing on main.sharelane.com. After you filed that bug into the bug tracking system, the developer fixes it and checks that code into the CVS. The build script picks up that new code as part of the new build and pushes that build on main.sharelane.com, replacing the previous build. Now you can verify if the fix is good or not.

Here are several things to remember about scheduled builds. Let's say we have the build script creating and pushing a new build to the test environment every three hours: 12:00, 15:00, etc., and it takes fifteen minutes or less to run a build script from start to finish.

- There is no sense in doing testing from 12:00 to 12:15, from 15:00 to 15:15, etc, because the build is being created and pushed, and when you do testing in the middle of the process, some files can belong to the previous build and some files can belong to the new one.

- If a programmer fixed your bug and checked in the fixed file(s) into the CVS, you'll be able to verify the fix only after the new build is pushed to the testing environment. So, if the checkin took place at 16:00, then your fix will be available for verification only at 18:15. Thus, in many cases it makes sense to launch the build script right when you need it. But if you do this, please **make sure that the other testers know about it** so you won't mess up their testing. In light of this, it's a good thing when each tester has his or her own testing environment.

BTW

It's a good idea to ask your release engineer to create a **build status page** where you can see

- Build id,
- DB version,

- Success status of build script run,
- Time when build script run was complete.

SL

See ShareLane build status page here: Test Portal>Release Engineering>Build Status

In some companies, release engineers create a Web interface to enable testers to push new builds when needed without the involvement of the release engineer.

Build numbering starts with 1 for a concrete release and increments by 1 every time a new build is created. So after we created the first build for the minor release 23.1, the unique build identifier called **build id** will be 23.1-1. After a new build is created, the build id will be 23.1-2, and so on.

BTW

As a rule, DB schema versioning is not linked to release versioning, so we don't start over with 1 with each new release like we do in the case of builds. We just increment each DB version by 1 every time the DB schema is changed.

At ShareLane, we specify the DB version after "/" following the build id; e.g., in 23.1-2/78, 78 is the DB version.

Before you start testing or bug fix verification, make sure that you are testing the correct application version by checking:

- **the build id** and
- **the DB version**

As you already know, you should ask the release engineer to provide you with an interface to easily identify the application version.

Using Python, Billy wrote the build script and added it to cron, so we have a new build being pushed to main.sharelane.com every three hours. The HTML code of each Web page of sharelane.com has the application version, so we just need to view the source of any Web page to know the version of the whole application. No more guessing and miscommunication about application version!

Finally,

- the code is written
- the testing is finished, and the bug fixes are made and verified
- acceptance testing is finished
- at our Go/No-Go meeting we decided that **we are ready for our first major release 1.0. Hooray!**

Our first live application version will be **1.0-23/34**

Here is what we must do to release 1.0:

1. Configure the production machine, e.g. create needed directories: /var/www/prod/cgi-bin/, etc.
2. Upload the SQL procedure to the production machine and run that procedure against the DB to create the DB schema with version 34.
3. Configure the build script to create the build on the production machine.

BTW

The production machine is simply a remote physical computer located at our hosting provider. That machine has a unique (among all computers on the Internet) identifier - an **IP address**, also called an **external IP** address. The format of the IP address is <0-255>.<0-255>.<0-255>.<0-255>.

BTW

If you want to find out the IP address of some server, do this (instructions are for Windows OS):

1. Click button “Start”.
2. Select option “Run”.
3. Type “cmd” in the dialog box and press “Enter”.
4. When the command prompt is invoked, type this:

ping www.google.com

Note that “http://” is not needed.

The value after “Reply from” is the IP address of one of the production Google machines. If we were within the sharelane.com Intranet, we could find out the **internal IP** address of the Star by typing:

ping star

or

ping main.sharelane.com

or by typing any other hostname of a Web site on the Star; e.g., billy.sharelane.com after “ping”

The difference between an external IP address and an internal IP address is that

- an **external IP** address must be unique among all computers on the Internet. An external IP address is like the address of an apartment; it must be unique among all apartments in the world, otherwise mail cannot be sent there.

- an **internal IP** address must be unique only among all computers on an Intranet. An internal IP address is like the conference room within a company; it must have a unique name among all other conference rooms within the company, otherwise we’ll be confused about where the meeting is: Is it in the conference room “Infinity” located next to the espresso machine, or in the conference room “Infinity” located next to the printers?

4. Run a build script to create a build on a production machine. The build script checks out files of the application version that we are going to release from CVS and copies those files to the production machine.

BTW

At ShareLane,

- Linux utility **scp** is used to copy files between Linux machines.
- Window utility **WinSCP** is used to copy files between Windows and Linux machines.

Both utilities are free. *You can find URLs to all the utilities mentioned in this Course under Downloads on qatutor.com, or you can just google their names.*

As our project evolves, our production environment will turn into tens of servers, which will form our **production pool**, but for now:

Ladies and gentlemen, our first release of sharelane.com is LIVE!

It's time for champagne, dancing on the desk of the CEO, reconciliation, forgiveness, and the realization that all those sleepless nights in the office were worth it.

Guess what? Users seem to like us! Our user base grows like crazy, and now we have hired two PMs, four more developers for the application core, one UI designer/developer, one DBA, one tester, and one CS (customer support) person. After another three weeks of hard work, we release version 2.0! But, once we poured the champagne to celebrate 2.0, our CS Nina bursts into the conference room and screams that she's been getting tons of complaints from users, because 2.0 is as saturated with bugs as the U.S. Congress is with lobbyists for a military industrial complex.

Basically, we have two options:

- Push 1.0 to prod, i.e. revert back* to the good old version.
- Fix bugs in 2.0 and push patch release 2.1 to prod.

* *another term is "rollback"*

It might look like the **first option** is problematic, because at ShareLane nobody seems to remember the exact version of each file for the production version of 1.0. Our build script is primitive, and as we create each new build, **we don't save the association between the build id and the file versions that belong to that build**. But after a couple of beers, Willy declares that first option is doable, because we can recreate 1.0 by checking out from CVS file versions **dated right before 1.0 was released**.

BTW

Association between the build id and files with their versions can be created inside a special log file. Every time we have a new build, the build script appends that text file with detailed information about that build.

Here is an example of file format:

<build id>;<DB version>;<success code: 0 - success (build is fine), 1 - failure (there were build errors)>;<filename/file version,filename/file version, etc.>;Unix timestamp.

That how that file might look like if we had only 2 files: register.py and checkout.py:

```
1.0-22;32;0;register.py/1.12,checkout.py/1.8;1206612924  
1.0-23;34;0;register.py/1.12,checkout.py/1.9;1206623816
```



See, build log of ShareLane here: Test Portal>Release Engineering>build_log.txt

In the general case, it's not easy to figure out if the second option (bug fix) is more or less attractive than the first one (rollback to old version), but this time it was easy for us. The reason is that we HAVE TO go for the **second option**. Why? Because we've just discovered that our DB procedures ... have not been checked in into CVS and nobody remembers the exact DB schema used for 1.0. So, even if we get the files for 1.0, there is a fat chance that they will be incompatible with the DB schema for 2.0 and thus we might get lots of bugs.

Example

register.py (1.0) queries table **users** (DB schema for 1.0), but if DB schema for 2.0 has this table renamed into **customers**, register.py (1.0) will not work.

Here are the negative consequences that immediately come as result of the failed release 2.0 and the decision to fix bugs for it.

- The programmers who fix the bugs for 2.0 don't work on 3.0.
- The programmers who don't fix the bugs for 2.0 cannot do a CVS checkin for 3.0, because it was decided to lock the CVS and allow only bug fixes for 2.0 to be checked in.
- The tester is spending his or her time verifying bug fixes for 2.0 instead of writing test cases for 3.0.

After the 2.1 patch release where all the nasty bugs have been fixed, we have a meeting in which Billy suggests that we have to take our version control to the next level by creating **branches** in CVS. He says:

“Okay, guys and gals. I have a four-year-old son Edward, and I have to send photographs of him once a month by email to my mother-in-law who lives in Rome. If a photo shows that Eddie is sick, then she calls and screams with anger, just as if she's used our 2.0. So what I do is this: I save photos of Edward looking good in a special folder, and if my mother-in-law starts to complain after getting a photo where Edward doesn't look healthy enough, I just say to her, “Wait a minute, that was the wrong photo,” and I email her a photo from my golden reserve of alive-and-kicking Eddies.

“Here is the story of our project from the release engineer's point of view:

"One day we started to write our code, and as we proceeded further, we decided to use CVS to store versions of our files. *At one point, we said 'Stop'* and decided to call whatever we had in the CVS 'version 1.0.' Then we started to add and checkin to CVS our new files and checkin into the CVS new versions of existing files, and again, *at one point, we said 'Stop'* and decided that whatever is in the CVS must be called 'version 2.0.' We did everything right, except one thing: the **files of 1.0 and 2.0 got mixed up because we didn't separate them.**"

"Now, imagine a tree: a trunk and branches."

Here is what we should have done from the beginning:

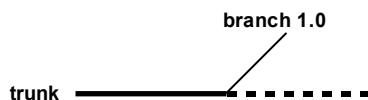
- The files created for and up to the 1.0 release make up the trunk of a virtual tree in the CVS.



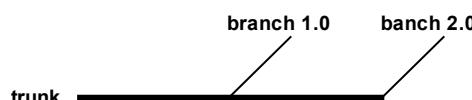
- Once we say, "Stop" for 1.0, we create (or "cut") a virtual CVS branch from the trunk, and that branch will contain our files for release 1.0 (the trunk will have those files, too).
- So now we have a CVS trunk and a CVS branch 1.0.



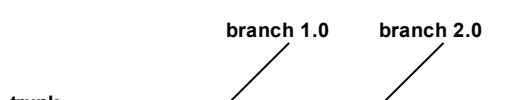
- The programmer who writes code for 2.0 must check in his or her files into the trunk (dotted line).



- Once the 2.0 code is finished, we cut another branch called 2.0.
- Now we have a trunk, a branch 1.0, and a branch 2.0.



- What shall we use to add/checkin the files for 3.0? Of course, the trunk (dotted line)...



- and so on.

This way, the code of each release lives in its own branch, or exists as a continuously updated trunk.

There are a lot of nuances about branching, but for now it's important that you grasp the concept of why branching is necessary.

What about our stuff? What's done is done. In our messy CVS, we have:

- all code for 1.0
- all code for 2.1
- part of the code for 3.0.

Let's call the trunk whatever we have in the CVS now. I'll spend my time finding all the files in their versions for 2.1, and I'll create a branch for 2.1, so if we release a buggy 3.0, we can easily go back to 2.1 by checking out the files from branch 2.1 and sending them to prod. And of course, from now on, we cut separate branches for **each** release.

And, last but not least, I'm going to fix our build scripts to

- enable logging associations between
 1. The build number;
 2. The file versions in that build;
 3. The time when the build is pushed to the target environment.
- enable the build script to create **any past build** when the build id is provided as input.

And ... I'm personally going to kick in the butt anyone who modifies the DB schema and doesn't check in SQL procedures into the CVS."

Let's thank Billy for his great presentation, and look at the benefits of implementing his suggestions:

First, we can easily get back to any of the previous versions.

Second,

- our programmers can concurrently work on as many versions as needed; e.g.; Billy can work on fixing bugs for 2.0 (branch 2.0), and Willy can work on writing code for 3.0 (trunk).
- the results of their work on each of the versions will be separated in CVS by the branching mechanism.

Third, we can control the state of each branch and trunk. Let's set up our branching mechanism to be able to have **three states of branches**:

OPEN: we can add/delete/checkin files (to/from/into CVS) without getting approvals, meeting certain conditions, etc. The trunk is always open.

CONDITIONALLY OPEN: we can add/delete/checkin files *if* we meet certain conditions depending on concrete situations. For example, in some companies approval from the dev manager is needed to add/delete/checkin files if a bug is found during acceptance testing, i.e. at the end of stage "Testing and bug fixes".

LOCKED: this applies to all branches with past/present production versions.

Let's look at the relationship between the stages of the Cycle and the openness of the branches:

1. During "Coding," the trunk is **OPEN**; the developers who are working on the coming release can mess up the trunk as much as they like.
2. During "Testing and bug fixes," the branch is **CONDITIONALLY OPEN**; the developers can do add/delete/checkin operations in the coming release branch only if they provide a valid bug number during actions with the CVS.

Example

When the developer tries to commit some file into conditionally open CVS branch, the CVS opens up a special text file, and the developer must type the valid bug number on the first line of it:

```
8766  
CVS: -----  
CVS: Enter Log. Lines beginning with 'CVS:' are removed automatically  
CVS:  
CVS: Modified Files:  
CVS: register.py  
CVS: -----
```

8766 is a bug number. When the developer saves this text file, a special program (a CVS "trigger") queries the DB of the bug tracking system whether two conditions are met:

1. A bug with this exact number exists.
2. That bug is in an open state.

If it's a double "Yes," then CVS allows the developer to save this text file and executes the original command passed to it. If one or both of these conditions are not met, then CVS asks the developer to enter a valid bug number.

3. Once the code is in production, then the corresponding release branch is **LOCKED**.

BTW

When a bug is found in production, the following situation can occur:

The developer

- checks in his or her fixed code into the branch with the patch release and
- forgets to checkin the fix into the trunk.

The consequence of that forgetfulness is this: **When the next branch is cut from the trunk, that new branch will have the same bug.** That's why we can have a situation when a bug that has been fixed in a previous production version (e.g., 2.1) reappears in production after a major release (e.g., 3.0).

So the golden rule is to **create a test case for each bug found in production**. Add this test case to a special test suite called "Test Cases for Production Bugs." I recommend keeping that suite dynamic: you add new test cases as production bugs are found (and fixed) and retire test cases from it once you execute them for the coming release and the next release after the coming release.

BTW,

When we encounter a bug in production, it's a good idea to have a **postmortem**. This term is borrowed from the medical field where it refers to a "medical procedure that consists of a thorough examination of a corpse to determine the cause and manner of death and to evaluate any disease or injury that may be present" (Source: Wikipedia).

By way of analogy, during a bug postmortem at a software company, we:

- do a thorough examination of why that bug was missed;
- try to identify weak points in our Cycle.

Depending on the severity of the situation, a postmortem can be held as a separate meeting or just as an email thread.

Postmortems should not be witch hunts. On the contrary, they should be constructive, positive measures targeted at improvements.

Sometimes Internet companies make a **beta release** prior to a major release. The idea behind a beta release is this: **Before we make an *official* major release (in other words, a major release available to ALL possible users), we make the code of that major release available to a limited group of people (beta testers) who represent our target users.**

BTW

A target user is basically a person who we expect to use our Web site. Target users can be identified by different sets of criteria: age, gender, occupation, interests, country, etc.

Beta testers are not test professionals. They are just regular folks that can be useful to us; e.g., we at sharelane.com can invite our most active users to be our beta testers if we decide to do a beta release. We can just send them an email with a secret URL; e.g., <http://beta.sharelane.com>. You can tempt users to become beta testers by offering them free items like t-shirts with your company logo.

We need beta releases, because beta testers will use our application in the same fashion as target users, and during beta testing:

1. Beta testers will report bugs to us.
2. We'll monitor our system and see how it works under real life usage. *For example, if the DB crashes during beta testing, we can assume that it will also crash after a major release when many more users are going to use that code.*

As beta testing goes on, we fix bugs and deal with other discovered problems (e.g., we might decide to add more servers to improve Web site performance). An example of a beta release is the email service Gmail: until Feb. 2007 new accounts could've been created by invitation only.

BTW

Please note that in some cases, a company will push out a major release with a label of “Beta.” So, if you see “Beta” on some Web site that’s available to EVERYONE, you can translate the word “Beta” as “This software is freshly baked and probably buggy. So don’t blame us if something wrong happens. Just send us an email with a description of the problem.”

As a rule, companies use beta releases in two cases:

1. The very first release (1.0) of the software.
2. The release of a large, important project; e.g., Gmail by Google.

The logical question is: “If we have beta testing, then we must have done alpha testing, right?” Yes, **alpha testing is the testing done BEFORE releasing the software to beta or regular users**; e.g., the testing done during the stage “Testing and bug fixes” is alpha testing. *Please note that alpha testing is performed by anyone inside the company who tries the new code before it’s released. For instance, the PM can ask the developer to play with the fresh code on the developer’s playground to see how the ideas from the spec are implemented in the software.*

Let's proceed.

Testers in Internet companies are in a privileged position compared to testers from other industries. If we at sharelane.com accidentally release a bug on production, we can do a patch release and remove the production problem within minutes. In many cases, that patch release will have a very low cost, and users will have no idea that a bug ever existed. But what if a P1 bug is found in the braking mechanism of an automobile?

- It will cost the auto company millions of dollars to make a recall.
- It will require **active user participation** to drive to the dealership to fix the problem.

Some last thoughts:

- A release that doesn't have a critical urgency must be pushed to production while the majority of users are nonactive; i.e., during the night. You can define a “night” for your releases using some interval of time (for example, from 00:00 to 6:00) in the time zone where most of your target users live. *This can be very difficult for Web sites with a big international exposure, like www.google.com.* As a rule, U.S. companies make releases between 11:00 p.m. Pacific Standard Time (2:00 a.m. Eastern Standard Time) and 3:00 a.m. PST (6:00 a.m. EST), so they have a four-hour window when the majority of people who live in the continental U.S. are asleep.

- Right before and during the time the release to production is under way, put a polite message like this on the production homepage: “Sorry for any inconvenience. This site is under maintenance. We'll be up at 3:00 PST.” It's not a big deal from a technical point of view, but your users will really appreciate your consideration.

- In many cases, a **coming release is not pushed to all the servers in the production pool, but rather to just one or a few of them.** The logic behind it is that we don't want to expose ALL of our users to the new code until we verify that this code works in real world conditions. So, random users hit our new code on one or several production machines and we monitor the quality in production by looking at the DB and log files. This approach is especially good for architectural releases when the front end is absolutely the same, but the back end is different.

- Depending on the specifics of the business, **Internet companies usually can predict the times when users are going to be more active than usual.** For companies that sell consumer goods, like Amazon, the period between December 1st and December 24th (the Christmas season) is the hottest time of year when a great deal of sales are made. If we know about that period of time beforehand, we must introduce a **moratorium for any release to production**, except EFB and EFR releases. The reason is simple:

- Our users need uninterrupted service.
- We don't want to jeopardize our major revenues.

Maintenance

Maintenance is about customer support, minor releases and other measures to keep our production environment and business in general up-and-running. Maintenance is everlasting activity. That's why we don't put it as a separate block of the Cycle flowchart.

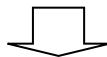
Now, take a deep breath. We are about to look at the Big Picture.

The Big Picture Of The Cycle

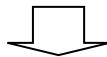
Let's use the The Simpsons family for our example. For our episode we need:

- Mother - Marge Simpson (PM);
- Son - Bart Simpson (programmer, tester, and release engineer);
- Father - Homer Simpson (user);

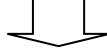
Idea: Marge says to Bart, "Your father will be happy if you'll build him a house just like this one in the picture as a gift for Father's day."



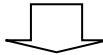
Product Design: Then Marge shows Bart an old photograph of Homer when he was five years old standing in front of his modest yet warm and loving house in Springfield.



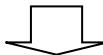
Coding: After his usual procrastination, Bart looks at the photo and starts to assemble Lego pieces to build a similar looking house.



Testing and bug fixes: When the house is ready, Bart starts to kick it, throw it against the walls of the Flanders house, insert/retrieve food into/from inside the house, and do other things that Homer would probably do. Each problem is fixed as soon as it's discovered.



Release: When everything looks fine with the house, Bart gives it to Homer.



Maintenance: Homer immediately tries to open a bottle of beer, and not being able to do this essential task, he asks Bart to fix the problem (*call to the customer support team*), which Bart does by gluing a bottle opener to the roof of the house (*patch release for EFR*).

Now, back to sharelane.com. First, let's recall our knowledge about players, their roles, and the stages in which they participate (note that the **activities for Maintenance can be performed at any stage**).

Player	Role	Stage
Marketing dude	Generate ideas and create MRD	Idea
Product Manager (PM)	Create spec	Product Design
Developer	Translate instructions from spec into software code	Coding
	Fix bugs	Coding Testing and bug fixes
Tester	Prepare test cases	Coding
	Execute test cases	Testing and bug fixes
Release Engineer	Push release	Release

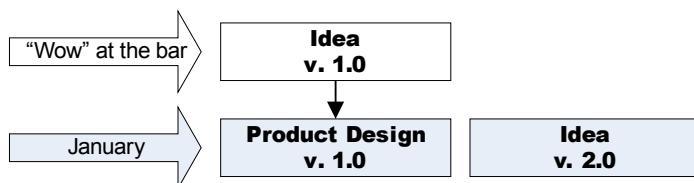
1. Let's start at the historic meeting at the bar when the **idea of v. 1.0** was conceived:



2. After we get an idea we have to develop a way to implement that idea; hence, we need a **product design for v. 1.0**. The product design is made by the PM.

At the same time,

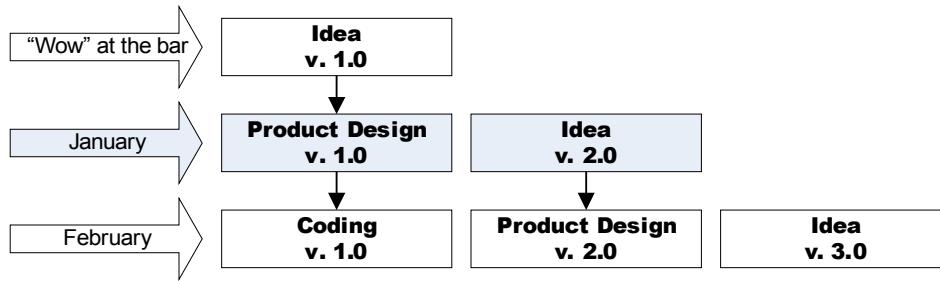
- the marketing dude generates **ideas for v. 2.0**.



3. After the product design for v. 1.0 is finished, we are at the next stage of the Cycle ("Coding") where the developer **writes code for v. 1.0**.

At the same time,

- the tester writes **test cases for v. 1.0**
- the PM creates **specs for v. 2.0** features
- the marketing dude generates **ideas for v. 3.0**

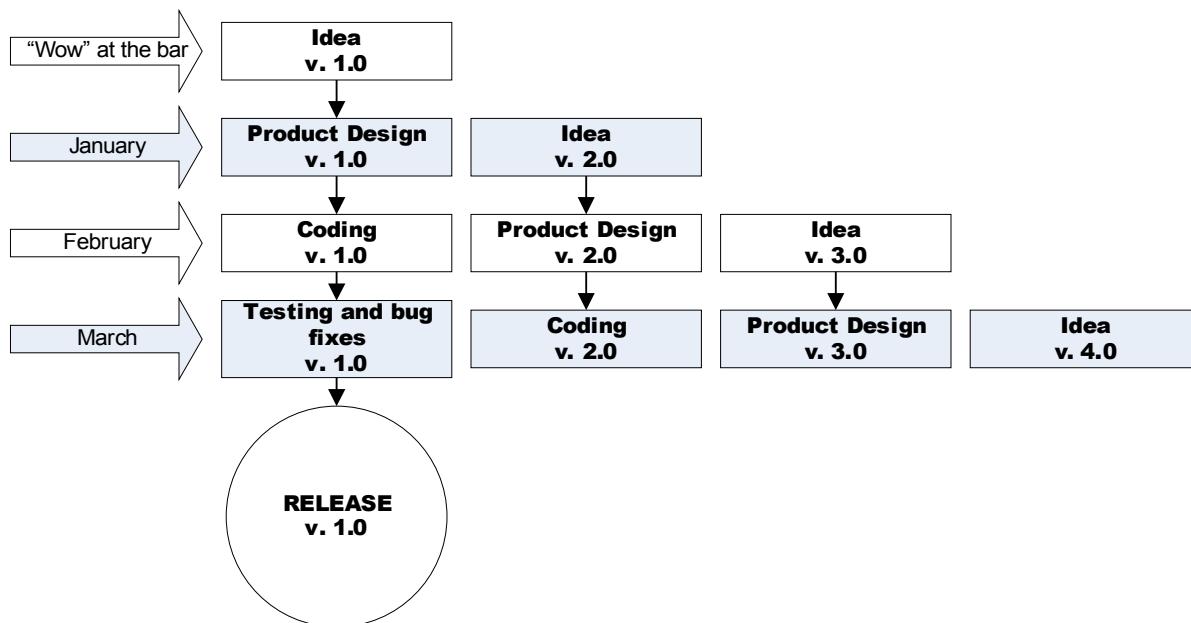


4. After the developer is finished with his or her code for v. 1.0, it's time for the testers to execute the test cases which have been written for v. 1.0 during the "Coding" stage. Now we are at the "**Testing and bug fixes**" stage of v. 1.0.

At the same time,

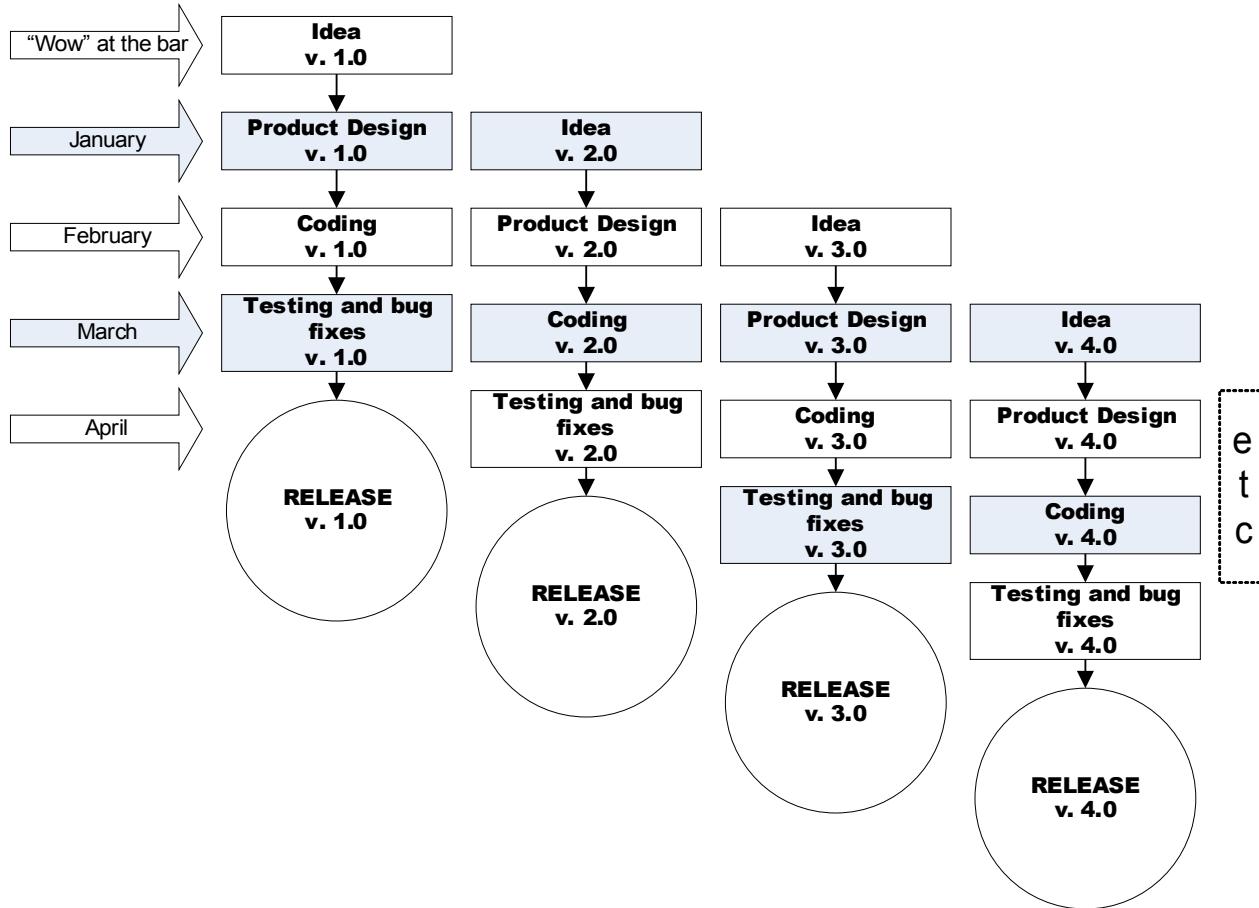
- the programmer writes code for v. 2.0
- the PM writes spec for v. 3.0
- the marketing dude generates ideas for v. 4.0

When the **Testing and bug fixes** stage is over, we have a major release of v. 1.0. Once v. 1.0 is pushed to production, the testers jump on the "Coding" stage of the Cycle for v. 2.0 and start writing test cases for v. 2.0 (see next page)



We have just looked at the Software Development Life Cycle for a major release version **1.0** of sharelane.com. After that, all is by analogy.

Now, let's look at the big picture (see next page):



BTW

The Big Picture is just a **model**; in real life (especially in the start-up environment) everything is not so smooth and structured. *For example, during the idea stage of v. 2.0, the marketing dude can generate ideas for both short-term (v. 2.0) and long-term (v. 4.0) projects.*

The main points to understand are:

- the concept of the Cycle
- *who does what* during each stage of the Cycle
- the concept that several Cycles can coexist. *For example, in March we have activities for four Cycles where the coming major release is 1.0.*

Lecture Recap

1. The **software development life cycle** is the process of taking a *desired software product* from the idea stage to the release of the *actual software* and its maintenance.

2. Below are main stages of the cycle:

- Idea
- Product design
- Coding
- Testing and bug fixes
- Release
- Maintenance

3. The **idea** refers to the purpose of the software or its parts. For example, the idea behind YouTube is to enable users to upload, share, and watch videos via the Web.

4. **Product design** is a description of how to practically implement the software idea(s).

5. The spec must have a unique ID and title. The spec must be assigned a priority. The spec's priority reflects the importance the company places on the spec's features. The highest priority is 1; the lowest priority is 4. Programmers and testers must always start working with specs that have the highest priority.

6. The PM should follow seven rules while creating specs:

- | | |
|---------|--|
| Rule #1 | Clarity of details and definitions. |
| Rule #2 | No room for misinterpretation. |
| Rule #3 | Absence of internal/external conflicts. |
| Rule #4 | Solid, logical structure. |
| Rule #5 | Completeness. |
| Rule #6 | Compliance with laws. |
| Rule #7 | Compliance with business practices. |

7. The spec goes through three stages: draft, approval pending, and approved. Only approved specs should be used for programming and test case generation.

8. Approved specs must be frozen and modified in accordance to the **spec change procedure**.

9. Examples, flowcharts, and mock-ups are excellent tools for illustrating specs.

10. One of the main challenges for the PM is writing specs that allow different people to interpret those specs the same way; i.e., exactly the way the PM intends the specs to be understood.

11. All internal documentation (including specs, test cases, code design documents, procedures, etc.) must be available on the Wiki and optimized for browsing and search.

12. **Coding** is the translation of the spec or requirements in some other form (for instance, a verbal request from a manager) into a software code.

13. The programmer who creates the code design document does a great favor to himself/herself and the company, because planning allows everyone to see future nuances and **prevent** bugs.

14. The most common architecture of a Web project consists of:

- The Web server
- The application core
- The database

15. Programmers develop code for the application core.

16. Below are some measures to assist companies to enhance best programming practices and **prevent** a substantial number of bugs:

- Hire good people, and be prepared to give them a break.
- Facilitate direct, fast, effective communication between coworkers.
- Conduct regular code inspections.
- Develop coding standards.
- Create realistic schedules.
- Ensure the availability of documentation.
- Set rules about unit testing.
- "If it ain't broke, don't fix it."
- Strive to create loyalty.
- Make "quality" and "the happiness of users" fundamental principles of the company philosophy.

17. The cost of the bug refers to how much money the company will lose, depending on when the bug was detected. In some cases when a bug is released to users, the loss to the company can ruin its business.

18. QA and testing should be regarded as business preservation mechanisms.

19. The 2 main activities performed by programmers are:

- Programming
- Bug fixing

20. There are 3 classic types of bugs associated with coding:

- Syntax bugs
- User Interface (UI) bugs
- Logical bugs

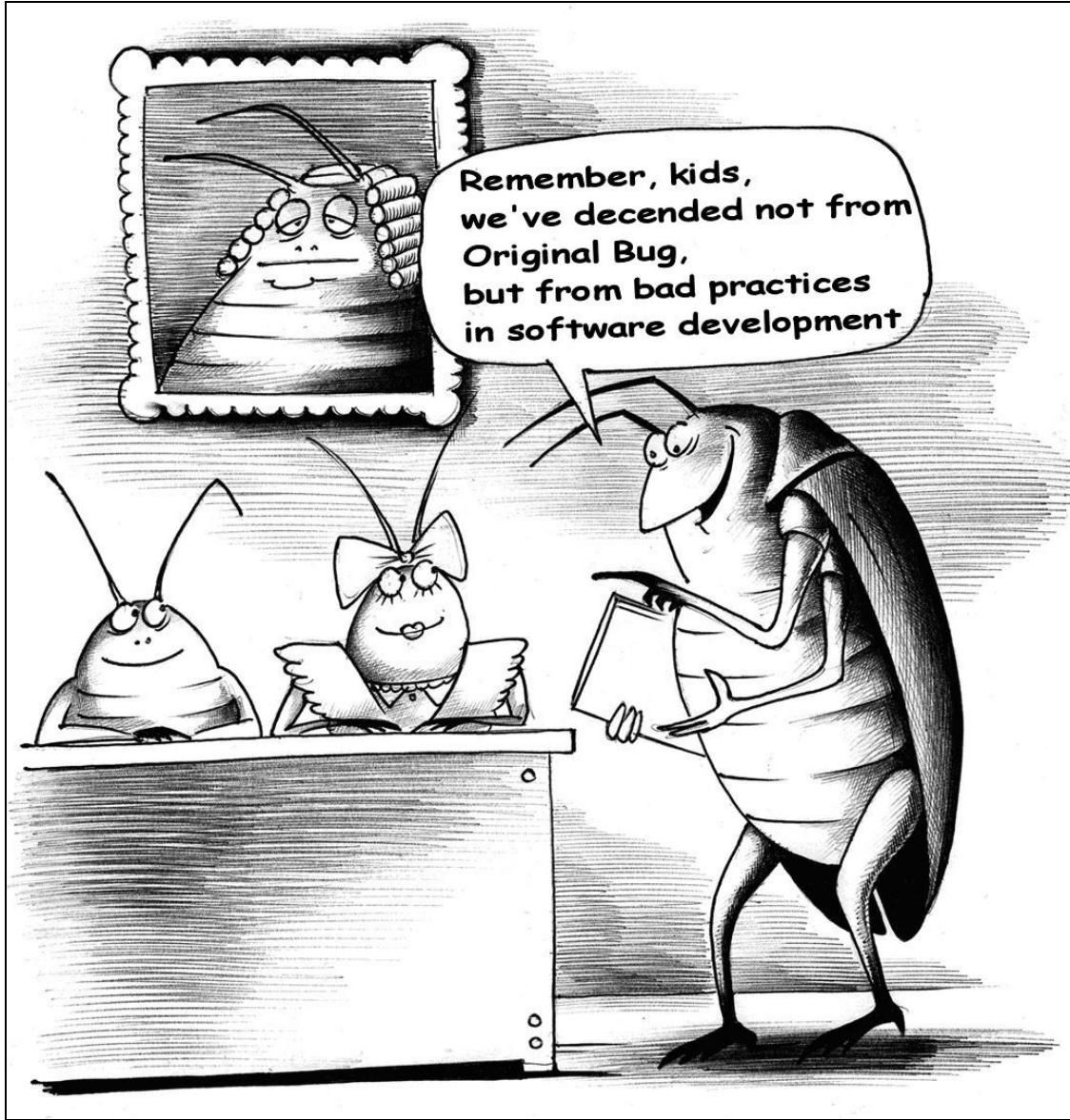
21. Syntax bugs are syntactic errors in the code. Syntax bugs are caught by the compiler or interpreter.

22. UI bugs are bugs in presentation.

23. Logical bugs are in processing.

24. Before you start testing, you must check:

- If the code is frozen
- If the test environment has the exact application version that needs to be tested



25. Software code, specs, and test suites must be stored in the CVS (or another version control system)
26. Test case reviews are an extremely useful practice, because they allow PMs and programmers to evaluate and provide feedback about how testers plan to test the software.
27. At the beginning of the "Testing and bug fixes" stage, testers must do a smoke test to check testability of the application. By way of analogy, you cannot test drive a car if the ignition doesn't work.
28. The 2 main stages of test execution are:
 - New feature testing
 - Regression testing

29. **Feature** is a broader term than **functionality**. **Feature** refers to:

- The ability to accomplish a particular task
- A specific characteristic of the software

30. Acceptance testing completes the Testing and bug fixes stage and is followed by a Go/No-Go meeting, in which a decision about the release is made.

31. The release is about making product available to the users. In case of beta release, product is made available to the group of selected individuals. In case of public release, product is made available to the general public.

32. A major release is a planned event that generally includes:

- New features
- Bug fixes
- Various modifications to or removal of old features

33. A minor release is either a planned or an unplanned event. As a rule, a minor release is usually a patch release (a release with bug fixes). However, a minor release can also contain new features and/or modifications or removal of old features.

34. A major release is a stage in the company wide SDLC, whereas a minor release can take place during any stage of the SDLC.

35. Bug fixes and various modifications or removal of old features can be considered maintenance.

36. An Emergency Bug Fix (EBF) is a situation where a P1 bug is found in production, and we need to push a patch release ASAP.

37. An Emergency Feature Request (EFR) is a situation where we need to release a certain feature ASAP; e.g., in the case of a court ruling or to comply with a new law.

38. EBF and EFR releases are treated as patch releases. P1 bug must be filed in both cases.

39. A DB refers to containers (DB schema) and the content inside those containers (DB data).

40. CVS branching is an important mechanism that allows us to painlessly roll back to old versions of software and do concurrent programming for multiple releases.

41. A CVS trunk is always OPEN for modifications, but CVS branches are OPEN, CONDITIONALLY OPEN, or LOCKED, depending on stage of the Cycle.

42. Bug postmortems are needed to find out why bugs were missed during testing.

43. Beta releases are needed as marketing and QA measures for 2 reasons:

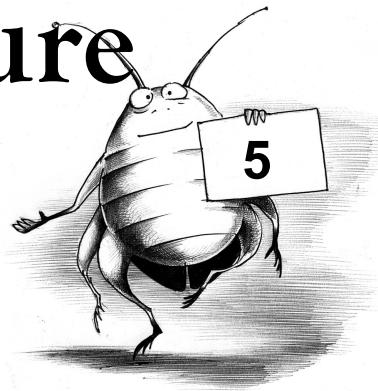
- To find out if part of the target audience is happy with the product

- To see how the system handles the activities of the beta testers
44. The release should be pushed to production while the majority of the users aren't using the Web site
45. In some cases, it makes sense to push a release to only one server of the production pool.
46. When there is high activity among users (i.e., seasonal items or the publication of a hot new book), it makes sense to set a moratorium for new releases.
47. Maintenance is an ongoing activity required to support the live site.

Questions & Exercises

1. List stages of the Software Development Life Cycle.
2. Which bug is more expensive: a bug caught during spec writing or a bug caught during test case execution?
3. List spec maladies.
4. Why shouldn't a PM give technical instructions in his or her spec?
5. Why do we need spec review meetings?
6. Why do we need spec freezes?
7. Why do we store our specs in CVS?
8. Why do we have bugs in our software code?
9. What is a unit test?
10. What is the code review, and how can it help us detect rascals who think that messed-up code is their job security?
11. Why do we need code freezes?
12. What kind of bugs can be detected by the compiler/interpreter?
13. What kind of bugs cannot be detected by the compiler/interpreter?
14. Why must files with test suites be stored in CVS?
15. Why is it beneficial both for the company and the tester to review test cases?
16. What happens if smoke test doesn't pass at the beginning of stage Testing and bug fixes?
17. What is the acceptance testing?
18. Why do we need Go/No-Go meeting?
19. What is the difference between a new feature testing and regression testing?
20. Give an analogy from real life to illustrate the word "release."
21. List types of releases and the differences between them.
22. If our coming release is 2.0, can we have some bug fixes for 1.0 in it?
23. If you answered "yes" to the previous question, why didn't we release a patch release after 1.0, instead of waiting until the next major release?
24. What does this version of the application mean: 34.2-56/212?
25. Why do we use version control software (e.g., CVS) for software development?
26. What is a CVS branch, and why do we need it?
27. List the states of the CVS branches and trunk.
28. Why do we need typical users to be our beta testers?

Lecture



The Software Testing Life Cycle

Lecture 5. The Software Testing Life Cycle.....	144
Quick Intro.....	145
Research.....	146
Test Planning.....	148
Test Execution.....	149
Test Education And Reality.....	150
Lecture Recap.....	150
Questions & Exercises.....	151

Before anything else, preparation is the key to success.

- Alexander Graham Bell

Practice is the best of all instructors.

- Publilius Syrus

Quick Intro

Let's have a quick, relaxed talk about the Test cycle while we cool down from our exciting (I hope) lecture about SDLC.

BTW

Remember that the SDLC has another name: the **software development process**.

Forget computers for a minute and imagine that you've just bought a super-duper, last-word-in-technology vacuum cleaner. How would you test it? After you remove it from the box, you'd probably read the manual and then torture poor thing until you're sure that it really works.

This wouldn't require too much thinking about the process, but let's look at the stages of that process on an abstract level:

1. **The instructions have been read and comprehended** (e.g., the description from item 22.1 of the manual on the wet cleaning mode).
2. **A plan has instantly developed in your brain.** For example:
 - a. Pour hot water into the upper water tank.
 - b. Press the "power" button.
 - c. Press the "pressure" button.
3. **The steps from the plan have been implemented** (i.e., you actually tried wet cleaning).

Let's jump from vacuum cleaner testing to software testing.

The software testing cycle (or "Test cycle") consists of three stages:

1. Research
2. Test planning
3. Test execution

At any of these stages testers can find a bug (i.e., a spec bug or a software bug). The bug must be fixed by the responsible party (the PM or the developer), and the fix must be verified by the tester.

Let's link the Test cycle with the SDLC:

1. **Research** starts before *spec approval* (which comes at the end of the Product design stage) and continues through Coding stage. Testers get their first look at the spec before the spec review meeting, and they keep researching the features they are going to test while writing their test cases.
2. **Test planning** takes place at the Coding stage when testers write their test cases.
3. **Test execution** takes place at the Testing and bug fixes stage when testers execute their test cases.

I've just described how the Test cycle is typically integrated with SDLC. The keyword here is "typically." In start-up reality, there WILL be a huge number of NOT typical situations.

Example

The test manager tells you that you have thirty minutes to test a certain feature (e.g., "Registration"). You have to use your common sense because there is no documentation and no time for inquiries about how it's supposed to work. In this situation, your research, planning, and execution all take place at the same time.

But for our course, I suggest a complete separation of the Test cycle from the SDLC. Here is another reason for this separation: When you perceive the Test cycle as an independent process, you can easily understand how it's integrated with **any** kind of SDLC in **any** software company.

To recap, the **independent process known as the Test cycle** consists of three stages:

1. Research
2. Test planning
3. Test execution

Research

The subject of the Research is basically **WHAT** is going to be tested.

Question: Why do people need software in general and Web sites in particular?

Answer: To satisfy certain needs, e.g., to buy books, read news, share videos, check credit card balances, etc.

Question: How can we, as software professionals, satisfy those needs?

Answer: We have to...

- **come up with** (marketing dude and PM),
 - **write** (programmer),
 - **test** (tester) and
 - **release** (RE)

...concrete means to satisfy those needs. These concrete means consist of **user-oriented features**. It's no surprise that most – or sometimes all – of the efforts made by testers are related to the testing of those features.

Brain Positioning

Testing user-oriented features is very important because these features are the reason users visit our Web site, and if those features are broken, the happiness of our users goes down, and, as we know, our profits are directly related to that happiness.

BTW

A logical question is: What features are NOT "user-oriented"? There are many other features that can be developed for these Internet projects. Those features can serve a variety of purposes: e.g., reporting, infrastructure, performance, etc.

For the sake of this course, we'll concentrate on **user-oriented features**. We'll simply call these "features."

Some examples of features:

- User can search book titles
- Link "Logout" is displayed only if user is logged in
- User can update quantity of books inside the shopping cart

What is the main source of information about features?

- Documentation (for example, specs, mock-ups, flowcharts, and other documents describing the features of our Web site)

An example of "other documents" could be instructions from our business partner about the file format for credit card transactions.

- Humans

Communication is an extremely important source of information. If you are in doubt about some of the spec provisions, feel free to approach the PM and ask as many questions as needed. In cases where there is no documentation and the features are not yet available for testing, your colleagues who have knowledge about the feature are the primary sources of information.

- The Web site itself

You can also learn about features by exploring previously written software: i.e., the Web site.

Let's discuss the concept of **exploration**. Each of us do this when we visit a Web site, fill in and submit forms, click links, and perform other actions that assist us to understand the Web site features. In Internet start-ups, exploration usually takes place in two situations:

1) When the code has been written, but the documentation is missing. Very often the first tester in the start-up finds himself or herself in this situation.

2) For self-education. When you come to a start-up company, they usually give you some time for getting started; everyone is extremely busy, and as a rule, nobody is going to babysit you.

What sources for expected results do we have during exploration? Primarily our old friends: common sense and life experience.

BTW

A very good way to better comprehend the features of the product is to become a regular user – i.e., go to the production environment, create an account, and do whatever other users do. Why? Because you'll take your understanding of the product to another level: **it's one thing to be a chef in the restaurant and another thing to be a customer.** It's a totally different perspective. Besides, you might get familiar with features that you've never used before.

The American expression "to eat your dog's food" applies to testers this way: "If your Internet company sells books, you should buy books from your company." It's also very useful to become a customer at your competitors' Web sites.

The main purpose of the **Research** stage is to answer two questions:

- Which features are we going to test?
- How are those features supposed to work?

When we have the answers, we can proceed to the next stage.

Test Planning

This stage requires lots of creativity and the full use of professional skills, because here many puzzles are solved in order to answer to **one very simple question**: "How are we going to test these features?" The quality of the product will be seriously impacted by the **wisdom** of decisions made here.

That wisdom is reflected two ways:

- a. Finding **short, simple, and elegant ways** to test our features
- b. Finding a compromise between
 - the volume of testing that's possible in *theory*
 - the volume of testing that doable in *real life*

The answers to this *one very simple question* are reflected in the test documentation. Just as the human body consists of cells, test documentation consists of test cases. Sometimes testers create additional pieces of test documentation (test plans, for instance, which we'll talk about later).

BTW

Sometimes test case generation goes along with using special automated helper tools that make test execution easy or, in some cases, make test execution possible. We'll cover test automation in a special section.

Test Execution



The gist of test case execution is conducting a **practical search for bugs in the code by using test cases that we've created during the previous stage.**

First, we perform new feature testing, using new test cases. As you recall, new test cases are often modified during their first execution.

Second, we perform regression testing, using old test cases.

During both new feature testing and regression testing, we try to find bugs. Remember:

- Once a bug has been discovered, the tester should file a bug report, entering it into the bug tracking system.
- After a programmer fixes the bug, the tester checks to see:

- a. If the bug was really fixed – i.e., the tester tries to reproduce the bug.
- b. If the activities during the bug fixing stage have unintentionally introduced new bugs – i.e., the tester does a quick test of the features that could have been affected

The testing performed during the steps a. and b. is also called **regression testing**. Thus, the expression "regress that bug" means that you have to perform the testing specified in those two steps.

Because the stages of **Research** and **Test planning** are intermingled, we'll unite them into one body of knowledge called **Test preparations**, or **Test preps** for short.

Therefore, the majority of our future lectures will be dedicated to two things:

1. **Test preps**
2. **Test execution**

Test Education And Reality

One of my main purposes for this course is to promote flexibility as one of the most important qualities of the start-up tester. In each particular start-up, some things (like processes or standards of test documentation) WILL work or look differently from what we've discussed.

In school, if you learned that $2+2=4$ and can be confident about the truth of that until you die, in testing you'll be constantly learning new facets about things you've been confident about.

Flexibility is **required** here! So, when I say that there are two stages in the Test cycle, you should be ready to experience a situation that might sound illogical at first but can easily take place in real life.

For example, you are given thirty minutes to test certain features, and after its release your manager asks you to create documentation for it!

Learn all you can, but **not** in a dogmatic way. Instead, try to understand the essence, and remember that in testing *nothing* is set in stone.

Lecture Recap

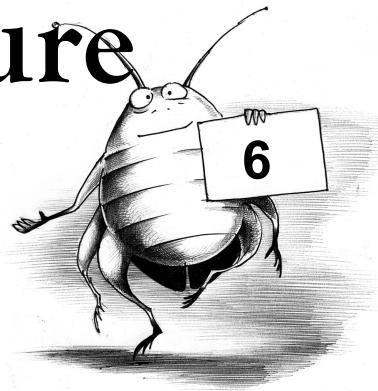
1. The Test cycle consists of two stages: "Test preps" and "Test execution"
2. During Test preps, we conduct research and write test cases.
3. The main sources of knowledge about a product's features are:
 - Documentation
 - Humans
 - Exploration
4. Good exploration skills require a quick understanding of how the software features work.

5. Test preps require great creativity and professionalism, because Test execution follows the testing ideas developed at this stage. Good testing ideas will assist us to find bugs.
6. Test execution consists of New feature testing and Regression testing.
7. Bug fix verification is also called *regression testing*.

Questions & Exercises

1. Why is it useful to perceive the Test cycle as an independent process from the SDLC?
2. What are the sources of knowledge about software features?
3. Why is it a good idea to become a regular user of your company's Web site as well as your competitors' Web sites?
3. What is exploration, and how does it help to fill the vacuum of information about the features?
4. Name the stages of the Test cycle.
5. Why do we test new features first?

Lecture



Classifying the Most Common Types of Testing

Lecture 6. Classifying The Most Common Types Of Testing.....	152
Quick Intro.....	153
By Knowledge Of The Internals Of The Software.....	154
By The Object Of Testing.....	162
By Time Of Test Execution.....	165
By Positivism Of Test Scenarios.....	166
By Degree Of Isolation Of Tested Components.....	168
By Degree Of Automation.....	176
By Preparedness.....	177
Lecture Recap.....	178
Questions & Exercises.....	181

The essence of the beautiful is unity in variety.

- William Somerset Maugham

The opposite of a correct statement is a false statement.

The opposite of a profound truth may well be another profound truth.

- Niels Bohr

Quick Intro

Any classification is based on certain criteria. For example:

- **By gender**, people are divided (classified) into males and females.
- **By having a cat**, people are divided into those who have a cat and those who don't.
- **By height**, people can be divided into groups based on how tall they are – e.g., one group from 150 to 159.9 cm, another group from 160 to 169.9 cm, and so on.

The same object can fall into an unlimited number of classifications. Take my friend Rajiv for example:

- Rajiv is a male
- Rajiv has a cat
- Rajiv's height is 165 cm.

Let's get back to software testing.

BTW

There are MANY types of testing, and I have no intention of presenting you with a comprehensive list. For this lecture, my purpose is to give you a general idea of the most common types. By classifying them, I'll simply help you to better understand and remember those types. I'd say that 99% of the test activities in software companies are the types that we'll cover here.

Here is the format:

<Criteria>

- <type of testing>

Let's go!

1. By knowledge of the internals of the software

- Black box testing
- White box testing
- Grey box testing

2. By the object of testing

- Functional testing
- UI testing
- Usability testing
- Localization testing
- Load/performance testing
- Security testing
- Compatibility testing

3. By time of test execution

- Before any release (alpha testing)
- After a beta release (beta testing)

4. By positivism of test scenarios

- Positive testing
- Negative testing

5. By degree of isolation of tested components

- Component testing
- Integration testing
- System (end-to-end) testing

6. By degree of automation

- Manual testing
- Semi-automated testing
- Automated testing

7. By preparedness

- Formal/documentated testing
- Ad hoc testing

By Knowledge Of The Internals Of The Software

- **Black box testing**
- **White box testing**
- **Grey box testing**

BLACK BOX TESTING

Imagine a soda vending machine. If you want a can of Coke, you insert money into the machine and get your soda. Nothing fancy. Now here's another perspective:

- The coins that you insert into the machine are INPUT.
 - The soda is an OUTPUT.
 - The buttons to select type of soda and the slot where you insert your coins are the USER INTERFACE (UI).
- The internals of the vending machine is like a **black box**, because you have a vague or no idea about the concrete mechanism that exchanges coins for soda.

In regard to software, the **black box (or area of unknown)** is simply a software back end. From a black box tester perspective, the back end can be thought of as a **virtual bridge** that connects INPUT and OUTPUT.

There are two main things about black box testing:

1. The tester usually has no idea about the internals of the back end.
2. Ideas for testing come from expected patterns of user behavior.

Let's look into the details.

1. The tester usually has no idea about the internals of the back end.

On the one hand, the tester has an advantage over the programmer, i.e., the author of the back end code. Why? It's human nature to see desired things as reality. **The programmer wants to see his or her code working**. Every parent thinks that his or her child is the smartest and the most talented. The code is the child of a programmer, and *in his or her reality* the programmer often perceives his or her code as a problem-free creature.

BTW

Here is my favorite legend about the power of perception:

When the ships of Columbus' expedition stopped in front of one of the Caribbean islands, the natives didn't see those ships because their perception did not include images that were completely different from what they and their ancestors had been observing for thousands of years. Only their priest felt that something was wrong, and he kept scanning the horizon until he could distinguish the silhouettes of the Spanish frigates from the sea and sky. Then he said, "Check it out, dudes: Columbus with his expedition," and only after that could those natives see reality as it was.

The black box tester has no "bonds" with the code, and a tester's perception is very simple: **a code MUST have bugs**. Using the principle, "Ask and you shall receive," black box testers find bugs where programmers don't.

BUT, **on the other hand**, black box testing is like a walk in a dark labyrinth without a flashlight, because the tester doesn't know how the back end was actually constructed. That's why there are situations when

- A black box tester writes many test cases to check something that can be tested by only one test case.
- Some parts of the back end are not tested at all

Therefore, black box testing has the advantage of an unaffiliated opinion on the one hand and the disadvantage of blind exploring on the other.

2. Ideas for testing come from expected patterns of user behavior.

What we've been calling "steps" is a combination of 3 things:

- Actions
- Data and
- Conditions

combined together to achieve an actual result.

Example

SL Do this:

1. Go to www.sharelane.com.
2. Type word "expectations" into the "Search" text field.
3. Click "Search" button.

Steps 1-3 inclusively are **actions**. The text "expectations" is the **data**. If scenario assumes that book is in DB, then the **condition** is: *DB has data about book with word "expectations" in its title*.

Scenario is a combination of actions and data applied to software under certain conditions

The purpose of a scenario is to bring test execution to the point where an actual result can be retrieved and compared with an expected result.

Expected patterns of user behavior are simply scenarios that we expect will be (OR are already) taking place as users use our software.

Thus, we can expect patterns of user behavior regarding software that

- **will** be released in the future
- OR
- **has already** been released

to our users.

How can we figure out expected patterns of user behavior?

- a. We can take them from **the spec** – e.g., the PM can assume the some users will enter an email address with two "@" characters.

- b. We can figure them out by **exploring**. When you browse your Web site, you can imagine what a regular user would do.
- c. We can develop scenarios by using a **black box methodology**. We'll cover this extensively.
- d. Scenarios can be a gift from our **intuition**. You can just wake up in the middle of the night and think: "What if a user would do that?" IMHO, intuition is one of the greatest assets that software tester might have.
- e. The **PM or programmer** might give you some valuable ideas.
- f. There are many other sources. For example, you can read an article about how users interact with similar Web sites.

BTW

In some cases, **after** we have released the software we might discover some **actual** patterns of user behavior. For example:

- The developer who works on reporting can share with you the results of live data about how users are using the site.
- The customer support folks can give you that information.
- You can read customer feedback on Internet forums, blogs, etc.

We can use knowledge about the **actual** patterns of user behavior to:

- Change our existing test cases (that we created based on **expectations** about patterns of user behavior) and/or
- Add new test cases

We would use these changes and/or additions to increase the probability of bug finding.

Let's do a quick recap about conceptual issues regarding black box testing.

The black box approach assumes that the tester usually doesn't know how the back end was written, so ideas for testing come from expected patterns of user behavior. Expected patterns of user behavior are scenarios that we expect will be (OR are already) taking place as users use our software.

Here's another illustration of black box approach.

Example

Spec #2233: "5.1. If user adds from 20 to 49 books (inclusively) to the shopping cart, we will offer a 2% discount."



Now, do this:

1. Create a new account on ShareLane and login (you can use Test Portal>Helpers>Account Creator to create the new account and login automatically.)
2. Add any book to the shopping cart.
3. Click the "Shopping Cart" link.
4. Change the number of books to 20.
5. Click the "Update" button.
6. Check the value under the "discount" column.

Expected result: 2

We just performed black box testing by executing a scenario when a user adds 20 books to the shopping cart. The expected result was 2, but is the code really bug-free? Read on!

WHITE BOX TESTING

White box testing (also known as "glass box testing," "clear box testing," and "open box testing") encompasses a number of testing techniques that require a comprehensive understanding of the software code. For example, a programmer can perform white box testing by comparing:

- the requirements from the spec
- a piece of Python code from ShareLane

Do you want to perform some white box testing now? Let's do it!

Example

Spec #2233: "5.1. If user adds from 20 to 49 books (inclusively) to the shopping cart, we offer a 2% discount."

SL Now, do this:

1. Go to Test Portal>Application>Source code>shopping_cart.py
2. Click the "BUG #1" link under the "View bugs" section.

Here is what we see:

If $q \geq 12$ and $q \leq 49$:

discount = 2

Programmer Billy used "q" as a variable to hold the value of the quantity of books added to the shopping cart. In human language the expression means:

IF the quantity of books is greater than or equal to 12
AND it is also less than or equal to 49

THEN the discount should be equal to 2

Congratulations! We've just found a bug using the white box testing approach! Billy made a mistake by writing 12 instead of 20. What is the bug summary? "**Spec2233: shopping_cart.py: 12 instead of 20 was given as lowest limit for a 2% discount.**"

During black box testing we used "20" as an input and it was a legitimate scenario, BUT we didn't find a bug! We would have found a bug if we had tried "19" – i.e., a value that was not mentioned in spec. The bug was actually found during white box testing. That brings us to two very important conclusions:

1. If we simply develop test scenarios using direct ideas from the spec, we will create legitimate test cases, but those test cases won't necessarily be effective bug finders. **The job of the PM is to describe how the features should work, not how the features should be tested.** In other words, the PM develops USE CASES, not test cases.

Brain Positioning

A **use case** is a description of :

- How software will be (or is being) used
- How software should respond to certain scenarios

Example

Here is an example of a use case: "*During login, if a user submits the wrong password, an error message 'Oops, invalid username or password' should be displayed.*"

A functional spec often includes a collection of use cases.

Use cases and test cases are *similar*, because they share the same concept: a certain scenario should lead to a certain expected result.

Use case and test cases are *different* because:

- The purpose of a test case is to find a bug, while the purpose of a use case is to describe how the software should respond to a user actions.
- A test case is used to *test* the software, while a use case is used to *describe* the software.

You can read an excellent article about use cases on Wikipedia.

The real power in finding bugs is invoked when we use the professional body of knowledge known as the *black box testing methodology*. The value of "19" originates as a test idea because I used the special black box testing technique called *Boundary Values*. Read on, and soon you'll become experts in many cool testing techniques!

2. Black box testing and white box testing are a great combination that helps to find bugs by improving:

- **Test comprehensiveness** – i.e., checking the software from different angles
- **Test coverage**

Brain Positioning

Let's elaborate on test coverage.

Imagine a chessboard with 64 squares with white king on one of the squares. Each possible position of the king on the board is written on a separate card in the form of an instruction – e.g., "Move white king to E2." There should be a set of 64 cards to cover all of the possible positions. If we start moving the king using the positions specified on our cards, then, after all cards are used, we will achieve:

- 100% of the possible positions of the white king on the chessboard
- 100% execution of the given instructions

Now, imagine that the chessboard is so big that the number of squares is incalculable. Let's also imagine that, according to some logic, we selected 20 positions and wrote them down on 20 cards.

Can 20 cards cover 100% of the possible positions of the white king on the chessboard? No. But can we achieve 100% execution of the given instructions? Yes.

Now, back to software testing.

The term "test coverage" means one of the following based on the context:

- 1. The coverage of possible scenarios**
- 2. Test case execution coverage**

1. For the coverage of possible scenarios, we have two situations:

a. Usually, the number of possible scenarios is incalculable. Thus, we usually don't really know the exact percentage of possible scenarios that we are going to cover by our testing.

b. In some cases, the number of possible scenarios IS calculable – especially when we narrow down our testing to a small, isolated piece of the software. In those cases, as we prepare test cases we might confidently name the percentage of possible scenarios to be covered during our testing.

In both situations, the coverage of possible scenarios is improved if we add a **valid unique** test case – e.g., a test case that:

- Checks possible scenario
- Does not duplicate another test case

When we talked about how the black box/white box combo helps to improve test coverage, we also talked about improving the coverage of possible scenarios.

2. It's much simpler with test case execution coverage.

We always know how many test cases we have and how many of them have been executed.

So, when you are asked about test coverage, ask what that person means: the coverage of possible scenarios, or test case execution coverage.

In real life, white box testing usually exists in the form of **unit testing** performed by the programmer against his or her own code.

GREY BOX TESTING

This is my personal favorite, and I consider it the most effective approach that I've seen during my entire career. One of the main reasons why I created ShareLane was to provide you with a grey box testing experience.

Grey box testing combines the elements of black and white box testing.

- On the one hand, the tester uses black box methodology to come up with test scenarios.
- On the other hand, the tester possesses some knowledge about the back end, AND he or she actively uses that knowledge.

BTW

Do you remember Test Case with Credit Card? We did DB verification – i.e., we used knowledge of the back end for our testing. That was a typical example of the grey box approach!

Below are two things to conclude our quick intro to black, white, and grey box testing.

1. During black box testing, the tester's actions are NOT limited to actions that can be performed by the users. For example, ShareLane testers use Credit Card Generator (Test Portal>Helpers>Credit Card Generator) to generate a credit card for testing. Obviously, users don't have access to our internal test tools.
2. One of the most critical things to keep in mind while doing black, white, or grey box testing is to come up with expected results that serve as **true indicators** of whether the software works or not. Please pay attention to this.

Example

In the Test Case with Credit Card, we checked the DB value. We did that because the true indication that the credit card transaction was successful was reflected as a second digit in the result column of the table cc_transactions – e.g., in the case of a successful Visa transaction, the value must equal "10". Why is this a true indicator? Because if the transaction is successful, the credit card processor sends us a success code that equals "0". We can say, "We used the grey box approach," because we used DB verification.

If we used a pure black box approach, we would only be able to verify that, as a result of successful checkout, a user gets the confirmation page with an order ID. Will that confirmation page be a legitimate expected result? Yes. It IS an expected result that makes sense. But would it be a true indicator that the credit card transaction was successful? No.

Again, it does make sense to check if the confirmation page is displayed (in Test Case with Credit Card, we did that indirectly, because we got an order ID from that page), but a true confirmation of the success of the transaction can only be retrieved from the DB.

That's why I'm such a strong believer in the grey box approach – it provides more opportunities to find those expected results that serve as true indicators of whether the software works or not.

By The Object Of Testing

- Functional testing
- UI testing
- Usability testing
- Localization testing
- Load/performance testing
- Security testing
- Usability testing
- Compatibility testing

FUNCTIONAL TESTING

Functional testing is required to find logical bugs in the Web site functionalities. Test Case with Credit Card is an example of functional testing. We'll cover black box testing techniques for functional testing in a separate lecture.

UI TESTING

UI testing is required to find bugs in the user interface of the Web site.

You should understand the difference between:

- 1) UI testing
- 2) Testing using UI

In first case, you test UI.

Example

SL Here is typical test case for UI testing:

1. Go to www.sharelane.com.
2. Click the "Sign up" link.

3. Check the number of characters that can be typed into the "ZIP code" text field.

Expected result: 5

In second case, you test whatever you need and simply use UI as a means to bring the software to a certain state.

Example

SL Here is a typical test case where we just use UI to test logic of the code:

1. Go to www.sharelane.com.
2. Click the "Sign up" link.
3. Type "2008" into "ZIP code" text field and press the "Continue" button.

Expected result: error message that ZIP code must have 5 digits.

We *will* cover some topics about UI testing, but our main focus will be functional testing. Why?

- Functional testing is much more important and much more difficult than UI testing.
- Most software testing activities are about functional testing.

USABILITY TESTING

Usability testing is the evaluation of the user experience when he or she uses our software. Sometimes usability testing is regarded as part of UI testing, and sometimes it's not.

Example 1: Each of us have seen "genius" Web site interfaces that are like an intricate puzzle when you attempt to perform some basic action (like finding the store locator page on some pizza Web site).

Example 2: One of the reasons why folks prefer Facebook to MySpace is that Facebook has a neat, conservative look, while MySpace users often abuse their ability to change the look of their Web pages by producing ugly monsters that hurt your eyes by flashing pictures and white text on a yellow background.

BTW

In the blessed nineties when VCs were joyfully pouring millions into the bank accounts of freshly baked start-ups, it wasn't unusual for start-ups employees to go out onto the streets of San Francisco and beg strangers to do usability testing for 50+ bucks an hour. Those were the days, my friend...

Feel free to talk to the PM if you think that your Web site's usability sucks.

LOCALIZATION TESTING

Localization testing is required to find bugs in the **adaptation** of our software for users from different countries. For example, if our Web site was created for an English-speaking audience and we want to localize it for a Japanese-speaking audience, we'll have to check if Kanji symbols can be used to create a username.

LOAD/PERFORMANCE TESTING

Load/performance testing is a set of testing techniques required

- To check the response time of our Web site or its components
- In case of a problem with response time, to find out what the problem is

The response time of a Web site is usually about how fast a user sees the next page after he does some kind of interaction – e.g., submit a registration form.

The most common technique is to incrementally increase the number of users until some concrete benchmark (e.g., 4 seconds of response time) is broken. That way, we can approximately predict the maximum number of users that can use our site within a reasonable response time. In many cases, this testing is done to discover a **bottleneck**. A bottleneck is a part of the software or hardware that slows down the response time. In some cases, the bottleneck is just an SQL statement that consumes too much of the server's memory, so the resolution of this performance problem can be as easy as modifying that statement, or dividing it into two or more separate statements.

The most common way to deal with problems concerning response time is to add extra machines to the production pool.

SECURITY TESTING

Several years ago, one of my friends who had just arrived in the U.S. from Russia refused to use the Internet because she "was afraid of hookers." Every time she said this, we'd laugh like mad, because what she really meant was that she was afraid of "hackers," i.e., cyber criminals.

Cyber crime is a HUGE problem. Because of it, individuals and corporations lose billions of dollars every year. In his interview in the book *Founders at Work*, PayPal co-founder and CTO Max Levchin said, "*2000 was basically the year of fraud, where we were just losing more and more money every month. At one point we were losing over \$10 million per month in fraud. It was crazy.*"

Security testing refers to testing the protection against security breaches. In many cases, that testing takes form of simulated hacker attacks.

COMPATIBILITY TESTING

Compatibility testing is about checking the compatibility of **our** software with the hardware and/or software on a user's computer. For example, some projects might have special requirements for a user's video cards (hardware), so compatibility testing might be dedicated to checking how compatible the software is when used with different video cards. With Web-based applications, in the majority of cases, compatibility testing takes the form of cross-browser and/or cross-platform testing.

Example

Many years ago, during one project we found a bug where a certain version of OS "A" would restart if the image was uploaded to our Web site using a certain version of Web browser "B".

- When we tried to upload the image with a different version of "A" and the same version of "B" everything was OK.
- When we tried to upload the image with a different version of "B" and the same version of "A" everything was OK.

Therefore, the problem was that a specific OS+Web browser combo was not compatible with our Web site.

There are three conclusions here:

1. Compatibility testing involves a variety of third party hardware and/or software with which our software must be tested.
2. Compatibility problems are a reality, and a user might have a really bad experience because of those problems.
3. For a Web project, it's a good idea to have a test lab with computers that have different OS and Web browsers.

As you know, one of the sources of expected results is statistics.

- You can get statistics about the worldwide usage of OS and Web browsers if you Google it (e.g., "Web browser usage statistics"), or you can check out the links in the Statistics section under Downloads on qatutor.com.

- You can also ask your company's IT person to give you some stats about the OS and Web browsers of your company's users.

BTW

The IT person gets those stats from Apache logs. Apache logs have that info because the Web browser sends it as a part of the "user-agent" section of the HTTP header. You can Google the phrase "my user-agent" and discover how Web sites "see" your OS and Web browser.

Many developers and testers use Firefox, because it provides many useful plug-ins and generic features. But during testing, testers should follow the rule, "Eat your dog's food," and use Web browsers preferred by users. Whether we like it or not, for many years now the general audience has preferred Internet Explorer.

In my experience, compatibility problems mostly occur when a Web site uses some cutting-edge scripting technology (e.g., AJAX) that might be treated differently by different Web browsers.

By Time Of Test Execution

- Before any release (alpha testing)

- After a beta release (beta testing)

Alpha testing is done before any type of release. When you hear "alpha testing," it refers to the time **when** the testing is done, **not how** testing was done. As a rule, alpha testing is done inside the company. In some cases, alpha testing is outsourced to other companies. We'll be talking a lot about the types of testing techniques used by testers during the alpha stage.

Beta testing is done after a **beta** release. The value of beta testing is letting **some** representatives from our target audience try the product before we release it in the open.

By Positivism Of Test Scenarios

- Positive testing
- Negative testing

Example

Let's look at a simple scenario (**don't execute it, just read**):

1. Go to www.sharelane.com.
2. Click the "Sign up" link.
3. Enter <**some value**> into the "ZIP code" text field.
4. Press the "Continue" button.

There are two types of data that the user can supply as <some value>. The first type of data is **valid data**; i.e., 5 digits. The second type of data is **invalid data**; i.e., **anything** other than 5 digits.

A new user registration can continue ONLY if the user provides a 5-digit ZIP code – in other words, if a user executes a normal, error-free scenario by submitting valid data.

But in reality, a **user can make an error** and supply invalid data for the ZIP code.

Example

The file `checkout.py` uses the function `get_ccp_result()` to get the success code of a credit card transaction: "0" is success; "1" is a failure. The function `get_ccp_result()` connects to the server of the credit card processor, sends out certain information, and once the success code is received, `checkout.py` makes a decision whether the checkout is complete or not.

Now, imagine the following situation: a user enters valid credit card info and that card has enough balance for transaction, but `get_ccp_result()` returns 1 instead of 0. This can happen by number of reasons. One of them is that the connection to the credit card processor server is down. So, we have a situation where there is no user fault and **malfunction exists within the system**.

Definitions:

Negative testing checks situations that involve

- **User error** and/or
- **System failure**

Positive testing checks situation where

- **The software is used in a normal, error-free way** and/or
- **The system is assumed to be sound**

Below are some important points:

1. As a rule, **negative testing finds more bugs**. There are two main reasons for this:

a. Errors and failures can take many shapes and forms, so the PM and the programmer might not predict some of them, and thus the code will not be ready to handle certain abnormal situations. Leo Tolstoy put it this way: "Happy families are all alike; every unhappy family is unhappy in its own way" – i.e., abnormalities have a great diversity.

b. When writing and developing features, it is natural to concentrate on the normal usage and normal functioning of the software because that's how we provide value to our users. In other words, users don't come to ShareLane to see error messages; they come to buy books: books can be bought only if

- the user executes an error-free scenario **and**
- our system is working normally.

2. Negative testing involves more creativity and puzzle-solving than positive testing. This happens for the same reason as above: errors and failures can take many shapes and forms.

3. Both negative and positive tests must be performed as a part of functional testing, but the rule is that we must execute positive tests first. Why? If functionality doesn't work during normal usage, it doesn't really make sense to check if it works with abnormal usage. It's like checking to see if a corpse has chicken pox.

4. **Error handling** is:

a. **How the system responds to errors made by users**; e.g., how the system responds if a Web form is submitted with invalid data in a required field.

OR

b. **How the system reacts to errors that happen when the software is running**. For example, this error message: Test Portal>More Stuff>Python Errors>register_with_error.py provides info that the file register_with_error.py calls the undefined function get_firstpage().

5. **Error message** is a message that provides information about error(s).

An error message is an important measure that:

- **Guides users in case of mistakes.** An error message is usually delivered via a Web page by a code that is specifically written for handling user errors.

- **Gives debugging info to developers.** An error message is usually provided by an interpreter (or a compiler), or by some logging mechanism: e.g., the Apache Web server records errors in a special error log.

By Degree Of Isolation Of Tested Components

- Component testing
- Integration testing
- System (end-to-end) testing

Let's define first and then illustrate.

Component testing is the functional testing of a logical component.

Integration testing is the functional testing of the interaction between two or more integrated components.

System (end-to-end) testing is the functional testing of a logically complete path consisting of two or more integrated components.

Let's illustrate.

Programmer Willy was asked to write a code that would find the **full names and emails** of all users who spent more than \$1000 shopping at ShareLane. We are going to send those users an **email with a gift certificate** giving them a 5% discount on a single purchase. Those certificates will be able to be used through November 17th. This feature is called "5% discount".

BTW

Please note that I've deliberately chosen not to illustrate this section with existing ShareLane functionalities. I did that to give you a feeling about test planning for software that doesn't exist yet.

COMPONENT TESTING

Let's single out 3 components to test:

Component 1: Generation of certificate codes and the generation of a data file

Component 2: Generation of and sending emails

Component 3: Usage of certificate codes

Let's plan our testing!

Component 1

Program cert_generator.py has this logic:

Check DB for qualifying users

IF qualifying users are found:

- >Generate certificate codes and insert them into special DB table;
- >Generate data file with full names, emails and certificate codes.

ELSE

- >exit

Let's narrow down our testing right away: We are not even going to consider a situation where there are no qualifying users, because we know for sure that we have plenty of them in production.

So, as a result of run of cert_generator.py, we should be able to check 4 things:

- Data in DB
- Data in file
- Format of certificate code
- Format of data file

When the code is ready for testing, we'll perform testing on main.sharelane.com ("Main"). This test environment has its own DB, and, if necessary, we can manually change some data in it.

Test planning:

1. Login into the DB of Main and reset the total amount spent by each existing user to 0 (zero). We need this step to prevent legacy (existing) data from interfering with our testing.

Brain Positioning

Remember that when you change a global setting (e.g., the system time) or do global DB manipulation on a shared test environment, you should **ALWAYS** ask those who use the same environment if it's okay to make those changes. You don't want to mess up their work!

The need for global modifications for the purpose of testing is one of the reasons why it's a very good idea for each tester to have his or her own test environment on his or her own Linux machine. With the low prices of hardware and all the free software (e.g., Linux, Apache, Python, MySQL), this shouldn't be a financial burden for a company.

2. Create data for **positive testing** – i.e., create new user accounts and make them big spenders (those who spend \$1000.01 or more). I suggest that you create only two accounts (just trust me for now; you'll learn *why* during our talk about Boundary Values):

- One account that spent \$1000.01
- Another account that spent \$1002 dollars

BTW

As a shortcut, instead of creating new user accounts, you can pick two legacy users and manually update the DB with the necessary amounts – i.e., \$1000.01 and \$1002 – for those users.

Brain Positioning

You have to really know what you are doing when you hack into the DB – i.e., when you perform a manual modification to DB data. Even if your SQL query achieves its goal, you can end up with a situation where the data integrity* is seriously compromised.

Data integrity can be defined as the **correctness, completeness, wholeness, soundness, and compliance with the intention of the creators of the data (definition taken from TechWeb.com).*

When you make a manual modification to DB data, you almost always affect data integrity – the question is to what extent. In some situations, nothing will break, but in others bugs will show up in the most unexpected places, and the programmer will have to spent hours to debug the problem, only to realize that those were not real software bugs, but consequences of violated data integrity.

So, again: Know what you are doing.

3. Create data for **negative testing**. You need two accounts:

- One that spent \$1000
- Another that spent \$998

Here again, we can also do testing by using two existing accounts and hacking into the DB to update the table with the spending history for those users.

4. Run cert_generator.py.

Expected results about the data:

- **In the DB:** 2 certificate codes are generated, 1 for each big spender
- **In the file:**
 - >the file has 2 records: 1 for each big spender
 - >for each big spender: first and last name, email and the certificate code are the same in the file and in the DB.

BTW

In our case, the certificate code is a combination of 10 alphanumeric characters (i.e., letters and numbers) in uppercase (e.g., LKJHG61123).

BTW

In our case, the data file must have a **comma-delimited** format – e.g., data blocks that are separated by a comma. Here is the layout:

<first name><SPACE><last name>,<email>,<certificate code>

Here is an example of the file:

```
Ferdinando Magellano,f.magellano@trinidad.pt,QWERT98362
James Cook,james.cook@endeavour.co.uk,ASDFG54209
Ivan Kruzenstern,ikruzenstern@nadejda.ru,LKJHG61123
```

Expected results regarding the *format*:

- Certificate code

>It should have 10 alphanumeric uppercase characters. We can look in the file to check the format of these codes.

- File

>Each block must have the expected info. Look at the file format. The first block must have the first and last name separated by a space; the second block must have the email address; the third block must have the certificate code. Please note that we don't check to see if the record has info about the **qualifying** users – i.e., users who spent more than \$1000 (we've already tested that); we simply check the format.

>Blocks must be separated by commas.

Component 2

Let's test this component in isolation. We'll do that by creating the file manually. How do we create the file? Not a big deal; we can do it using a text editor like Notepad.

Test planning:

1. Manually create a data file and insert one record with the following data:

- A random first and last name
- A valid email, so you can check whether the software has REALLY sent the email with the correct content
- A random certificate code

2. Feed the file as input to cert_sender.py. This Python script

- Constructs email messages parsing the data file
- Sends out the emails

After cert_sender.py runs, there are two expected results:

- Email was received (positive testing)
- Email message has the correct content (positive testing)

Also, you can do negative testing:

1. Create several bad data files manipulating with absent and/or corrupt data, and/or wrong format. For example record has no email:

Ferdinando Magellano,,QWERT98362

2. Feed those files to cert_sender.py one by one.

Expected result: error message each time when bad file is used as input (negative testing).

Component 3

Let's test this component in isolation. How can we do this?

As you recall, a certificate code is a combination of 10 alphanumeric characters – e.g., LKJHG61123. The certificate code must be entered during checkout, and if that code is valid, the total amount* should be reduced by 5%.

At ShareLane, we offer free shipping on all orders, but if we had shipping charges the 5% discount would be applied to the total amount **before shipping.*

We can:

- Ask a programmer to generate some certificate codes for us
- Find out how to hack into the DB to manually insert some random certificate codes
- Write a helper script to generate those certificates

Test planning:

1. Get 11 valid codes. Let's assume that we have these numbers:

11111AAAAA for User 1
22222BBBBB for User 2
33333CCCCC for User 3
44444DDDDD for User 4
55555EEEEEE for User 5
66666FFFFF for User 6
77777GGGGG for User 7
88888HHHHH for User 8
99999IIIII for User 9
10000JJJJJ for User 10
11000KKKKK for User 11
no certificate for User 12

2. Set today as October 21 (or any date before November 17) and execute test cases from below.

BTW

How do we simulate October 21, November 17, or any other date? Depending on the software design, the date is usually taken as the server system time or as some value from the DB. Therefore, we just need to change the system time or value in the DB.

Again:

Make sure that it's okay with others if you make a global change on a shared environment.
Be sure that the data integrity will not be seriously impacted if you change the date in the DB.

Test case	Step	User	Cert. code	Gets 5% discount? (Yes/No)	Comment	Type of testing (Positive/ Negative)
1	1	User 1	11111AAAAA	Y	Normal case	P
2	1	User 2	22222BBBBB	-	-	-
	2	User 2	22222BBBBB	N	User should not get a discount using the code again	N
3	1	User 3	44444DDDDD	N	Cannot use cert. that doesn't belong to you even if code is valid	N
4	1	User 5	66666FFFFF	-	-	-
	2	User 6	66666FFFFF	Y	This is combination of positive and negative testing to see if code still works after another user tried to use it.	PN
5	1	User 7	88888HHHHH	-	-	-
	2	User 7	77777GGGGG	Y	This is a combination of positive and negative testing to check if a user can use his/her own	PN

					code after a previous attempt failed	
6	1	User 12	QWERTYUIOP*	N	No discount should be given for invalid code	N

* This cert. doesn't exist

- Before executing each test case, set time to date, as noted in **Comment** column.

Test case	Step	User	Cert. code	Gets 5% discount? (Yes/No)	Comment	Type of testing (Positive/ Negative)
7	1	User 9	99999IIIII	Y	Date: November 17	P
8	1	User 10	10000JJJJJ	N	Date: November 18	N
9	1	User 11	11000KKKKK	N	Date: November 19*	N

* October 21, November 17, 18, and 19 have been picked by using the Boundary Values technique that you'll learn soon.

We checked Components 1, 2, and 3 in isolation; in other words, we assumed that they are independent from each other. Now it's time to test integration between those components.

INTEGRATION TESTING

There are three points of integration:

Component 1 -> Component 2 (data file generated by cert_generator.py is used by cert_sender.py)

Component 2 -> Component 3 (certificate code taken from email message constructed and delivered by cert_sender.py is used to get 5% discount)

Component 1 -> Component 3 (certificate code generated by cert_generator.py is used to get 5% discount)

Test planning:

Component 1 -> Component 2

- Login into the DB of Main and reset the total amount spent by each existing user to 0 (zero).
- Create one big spender account (e.g., \$2000 spent) with legitimate email.

3. Run cert_generator.py.
4. Run cert_sender.py against the data file generated by cert_generator.py.
5. Check your email account. Expected result:
 - >Email was received
 - >Email has correct content (positive testing)

Component 2 -> Component 3 and Component 1 -> Component 3

6. Use the certificate code from the email to get a 5% discount. Expected result:
 - >Discount is granted (positive testing)

SYSTEM (END-TO-END) TESTING

A system test is a scenario to test a feature from start to finish – e.g., component1 -> component 2 -> ... -> component n. In our case, the system test can be done by executing steps 1 to 6 inclusively used for integration testing.

We looked at the system test last, but I recommend that you create one test case with a simple system test as the first test case of your test suite. Why? Because it will serve as an acceptance test for that particular feature. In some cases, it doesn't make sense to do component/integration testing if the system test fails.

Example

Let's say that you executed a system test as a first test case, and no emails have been received. You filed a bug and went to talk to Willy. Willy looks into the code and tells you that Component 1 has several major problems. Naturally, you'll skip component testing for Component 1 and start doing component testing for Component 2 instead.

BTW

Speaking of emails, let me share a couple of things with you:

Thing 1:

The email address consists of the following parts:

email alias
@
mail server domain
. . .
top level domain

Your work email will likely have an alias made up of the first letter of your first name plus your full last name – e.g., rsavenkov – so my email at ShareLane might be: rsavenkov@sharelane.com.

When we test Internet projects, we usually have to create hundreds of user accounts. The problem is that each user must have a unique email address. If you register a user with the email rsavenkov@sharelane.com on Main, you cannot reuse that email. What can you do? Should you create countless accounts on Gmail and Yahoo, or there is a simpler way? The simpler way is this: ask your IT person to tune up your mail server in such a way that mail with the following format comes through:

<email alias>+sometext@<mail server>. <top level domain>

So, if I want to create a new user, I simply use this email rsavenkov+test1@sharelane.com, and if this email has already been used, I try something like this: rsavenkov+test1new@sharelane.com, etc.

BTW, it's a good idea to use something descriptive for your test email aliases: for instance, rsavenkov+component1@sharelane.com.

Thing 2:

In some cases, our testing requires that we enter a unique email address, but we don't need to check if an email from the system was actually received by the recipient. I've noticed that some testers will enter a meaningful email address like my email qatest@gmail.com. Please don't do that; instead, use an email like this (with a valid format, but no chance that it's a real email): apsdofaspdioh@iasudfhasdfuhasi.com – this abracadabra is even faster to type than qatest@gmail.com! Here are three reasons for not using emails that might be real:

- It's not nice to send junk mail (your company can actually get sued for that).
- It's not secure to send emails that contain sensitive data.
- It's not good for your company's image when somebody receives a test email.

By Degree Of Automation

- Manual testing
- Semi-automated testing
- Automated testing

MANUAL TESTING

This is testing without the help of any test automation programs. The testing techniques for manual functional testing are core elements of black box testing methodology.

SEMI-AUTOMATED TESTING

This is manual testing done with partial usage of the test automation, usually in form of helpers. For example, you can automatically generate a new user account with Account Creator and manually add a book to the shopping cart. Test Case with Credit Card is typical example of semi-automated testing.

Talking about Account Creator, I want to give you quick tip about the test password.

BTW

It's a really good idea to have a convention for default password for test accounts within the company. As you know, ShareLane's convention is "1111".

Why is this a good idea? Sometimes you don't have time to create a new user account with sophisticated settings. In the case of ShareLane, all user accounts are the same. But in the case of sophisticated products like financial software, testers can spend a lot of time creating necessary user accounts of a particular variety. Therefore, in some cases you might want to reuse an existing test user account. The problem is that if you remember the username but have forgotten the password, it's not easy to recover it because passwords are usually encrypted in the DB. But if you have a password convention, you don't even have to guess what the password is.

Another situation when having a password convention looks attractive is an exchange of information – e.g., when you give the developer info regarding the user account, he or she won't need to remember the password.

As always, a warning: if you reuse an account that was created by another tester, make sure to get his or her permission.

AUTOMATED TESTING

This is testing completely done by running test automation tools. Simple to perform yet valuable, automated testing can be done with link checkers; e.g., Xenu Link Sleuth (you can download it from QATutor.com). We are not going to cover automated testing in our course, except quick intro to automation scripts during lecture on regression testing.

By Preparedness

- **Formal/documenting testing**
- **Ad hoc testing**

Formal/documenting testing is a planned activity that requires the usage of test documentation – e.g., test cases.

Ad hoc testing is done without any preparation. Ad hoc testing relies on knowledge of the software and/or common sense and/or intuition. Doing ad hoc testing, the tester just follows his or her heart trying to find bugs.

Ad hoc testing is usually used:

- As a smoke test
- As a measure for extra peace of mind on top of formal/documenting testing
- As exploratory testing when a new tester comes to a company where code is being written and should be tested ASAP
- When everybody in the company does testing before some big release
- In other cases where there are no test cases

Ad hoc testing often produces amazing results. Sometimes you just test software without any plan and come up with unusual scenarios that help catch nasty bugs.

Lecture Recap

1. We've learned about most common types of software testing. We used classifications to organize the knowledge.
2. Black box, grey box and white box testing are 3 approaches based on the knowledge of the internals of the system.
3. Black box tester
 - usually has no idea about the internals of the back end
 - implements ideas for testing that come from expected patterns of user behavior.
4. Black box testing has the advantage of an unaffiliated opinion on the one hand and the disadvantage of blind exploring on the other.
5. During black box testing, the tester's actions are NOT limited to actions that can be performed by the users. For example, testers can use helpers for semi-automated testing.
6. Expected patterns of user behavior can be **figured out** with help of:
 - spec
 - exploring
 - black box methodology
 - intuition
 - communication
 - other sources
7. Actual patterns of user behavior can be **found out** with help of:
 - reporting data
 - data from customer support
 - info from forums, blogs, etc.
8. White box testing (also known as "glass box testing," "clear box testing," and "open box testing") encompasses a number of testing techniques that require a comprehensive understanding of the software code. White box testing is usually done by a programmer against his or her own code.
9. If test case checks legitimate scenario, it doesn't mean that it's well-geared to find a bug. The real power in finding bugs is invoked when we use the professional body of knowledge known as the **black box testing methodology**.

10. Black box testing and white box testing are a great combination that helps to find bugs by improving:

- **Test comprehensiveness** – i.e., checking the software from different angles
- **Test coverage**

11. Depending on context, "test coverage" means:

- either the coverage of possible scenarios
- or test case execution coverage

12. In case of grey box testing

- On the one hand, the tester uses black box methodology to come up with test scenarios.
- On the other hand, the tester possesses some knowledge about the back end, AND he or she actively uses that knowledge.

13. During black box testing, the tester's actions are NOT limited to actions that can be performed by the users.

14. One of the most critical things to keep in mind while doing black, white, or grey box testing is to come up with expected results that serve as **true indicators** of whether the software works or not.

15. We also learned that:

Functional testing is needed to find logical bugs in Web site functionalities.

UI testing is needed to find bugs in the presentation of the Web site user interface.

Usability testing is the evaluation of a user's experience when he or she uses our software.

Localization testing is required to find bugs in the **adaptation** of our software for users from different countries.

Security testing refers to testing the protection against security breaches.

16. Problem called "incompatibility" may take place when certain hardware and/or software on client-side interact with our software, i.e., a Web site.

17. Load/performance testing is a set of testing techniques designed to load the system or its component(s) and then measure how the system or its component(s) react

18. The usual purpose of load/performance testing is to find a bottleneck; i.e., a part of the system or its component(s) that slows down response time.

19. Functionally correct code – e.g., certain SQL statements – can cause performance problems.

20. Alpha testing is done before any type of release. When you hear "alpha testing," it refers to the time when the testing is done, not *how* testing was done.

21. Beta testing is done after a beta release.

22. Positive testing checks situations where:

- The software is used in a normal, error-free way and/or
- The system is assumed to be sound

23. Negative testing checks situations involving:

- User error and/or
- System failure

24. Positive test cases must be executed before negative test cases.

25. Error handling is:

- a. How the system responds to errors made by users
OR
- b. How the system reacts to errors that happen when the software is running.

26. Error message is a message that provides information about error(s).

27. An error message is an important measure that:

- guides users in case of mistakes.
- gives debugging info to developers.

28. **Component testing** is the functional testing of a logical component.

29. **Integration testing** is the functional testing of the interaction between two or more integrated components.

30. **System (end-to-end) testing** is the functional testing of a logically complete path consisting of two or more integrated components.

31. You have to really know what you are doing when you hack into the DB or change some global setting, e.g., server time.

32. Comma-delimited format is popular format for data files. Blocks of data are separated by commas.

33. Manual testing is done without the help of any test automation programs.

34. Semi-automated testing is manual testing done with partial usage of the test automation, usually in form of helpers.

35. Automated testing is completely done by running test automation tools.

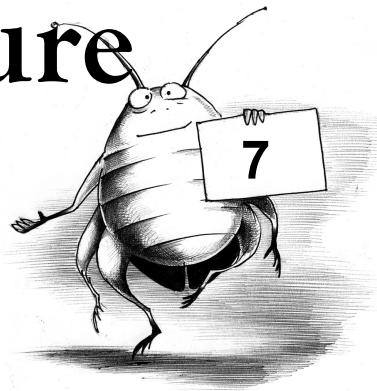
36. Formal/documenting testing is a planned activity that requires the usage of test documentation – e.g., test cases.

37. Ad hoc testing is done without any preparation. Ad hoc testing relies on knowledge of the software and/or common sense and/or intuition.

Questions & Exercises

1. What is the main difference between black box testing and white box testing?
2. What are advantages of grey box testing?
3. What is the "scenario"?
4. What is the "pattern of user behavior"?
5. What can be used to figure out expected patterns of user behavior?
6. What can be used to find out about actual patterns of user behavior?
7. Let's say we found certain pattern of user behavior. Does it mean that this pattern can be included into a test case? If "not", why not?
8. What are the benefits of using black, grey and white box testing techniques against the same piece of code?
9. What is the difference between coverage of possible scenarios and test case execution coverage?
10. Why is it so important to find expected result that serves as a true indicator of whether the software works or not?
11. What is the difference between functional testing and UI testing?
12. What is the difference between UI testing and testing using UI?
13. What is the bottleneck?
14. Can software code cause a bottleneck?
15. What is the difference between alpha testing and beta testing?
16. What is the difference between positive testing and negative testing?
17. What can be considered as normal usage of the system?
18. What is error-handling?
19. What is error message?
20. Why correctness of error-message have real importance?
21. What is the difference between component and integration testing?
22. Why it makes sense to execute test cases with system test first?
23. Why we should exercise real caution hacking into DB?
24. What shall we do before changing global setting on shared test environment?
25. What is comma-delimited file?
26. What kind of test automation is usually used with semi-automated testing?
27. Why in your opinion ad hoc testing often brings amazing bug finding results?

Lecture



Test Preps

Lecture 7. Test Preps.....	183
Quick Intro.....	184
The Tester's Mental Attitude.....	185
Intro To Special Skills In Bug Finding.....	186
Black Box Testing Techniques.....	187
When You Start To Implement Testing Techniques.....	210
Lecture Recap.....	210
Questions & Exercises.....	211

When I am working on a problem I never think about beauty.
I only think about how to solve the problem.
But when I have finished,
if the solution is not beautiful, I know it is wrong.
- Buckminster Fuller

I hear and I forget.
I see and I remember.
I do and I understand.
- Confucius

Quick Intro

From the tester's perspective, test preps can include:

1. The generation of new test cases
2. The modification of existing test cases
3. The deletion of existing test cases

Sometimes we also need to create and/or modify test automation helpers.

Brain Positioning

Many things that we'll be talking about will look simple in theory, but don't allow that pseudo-simplicity to mislead you!

Here is an analogy using chess. It takes 20 minutes and very little brainpower to understand the basic chess rules, but it takes years of practice and hundreds of actual games to become a chess master.

The same is true of test methodology: it will not be hard for you to understand it, but it will take lots of practice before your theoretical understanding will turn into practical value.

For a tester, test preps is the most complex, creative, and interesting activity in software testing. The final result of test preps is a set of test cases that are well geared to find bugs during test case execution.

Brain Positioning

The tester's job is to find and address bugs. But what if the test case execution result is PASS? Does it automatically mean that the tester failed to do his or her job properly? NO WAY! Imagine a fisherman who uses a net to catch fish. Even if the net is in perfect condition and it was used correctly, **sometimes there are simply no fish***!

* In a minute, you'll find out that we testers must always **believe** that there **are** fish... sorry... bugs.

But the bottom line is that **the net must be made well and used properly**.

Our net to catch bugs consists of our test cases. We weave/update the net during test preps and use it during test execution.

As you recall, a test case can consist of only expected result, but as a rule it's a combination of:

- Idea
- Scenario
- Expected result (-s)

The ideas for each of these 3 elements can be taken from many sources – e.g., specs, experience, common sense, etc.

Question: What is different about testers compared to the other participants of the SDLC who can also use specs, experience, common sense, etc., to generate test cases?

Answer: Testers are different because they possess two professional qualities:

- A special mental attitude towards software
- Special skills in finding bugs and addressing them

Let's talk about a tester's mental attitude first.

The Tester's Mental Attitude

People with different attitudes see things differently. Here is my favorite example: When 3 friends – a carpenter, a painter, and a biologist visited a forest and shared their observations, they found out that:

- The carpenter saw logs.
- The painter saw landscapes.
- The biologist saw material for an article.

Let's apply this principle to **software**:

- For the user, it's an instrument to solve specific tasks or to satisfy a need (e.g., to transfer money or to read the news)
- For the PM, it's a way to bring to life the ideas imprinted in the specs.
- For the programmer, it's a beloved baby.
- **For the tester, it's a shelter for bugs.**

Brain Positioning

The expression, "**Software has bugs**," is not a joke or an exaggeration; it's the Universal Cosmic Law. So, let's open our hearts to that Law! Let's have an unquestionable belief that:

- **It's the very nature of software to be buggy and unreliable**
- **If software seems to work, it's not normal – it means that something is wrong**

We've all heard these words of wisdom: "**Ask and you shall receive.**" As testers, we "ask" for bugs:

- ✓ By not trusting software
- ✓ By searching for bugs

The less we trust software, the more bugs we find. The more bugs we find, the less we trust software. Nice cycle!

The **lack of trust** towards software is connected with another tester's virtue: **destructive thinking!** We are not silent observers of this buggy substance called "software"; we actively find ways to demonstrate that the software is breakable. And what is the best way to show that the software is breakable? By finding bugs!

You can say, "*Wait a minute! During our previous lecture you told us about positive testing and how it must be executed first. How can positive testing coexist with destructive thinking?*"

Here is my answer: "*Positive testing is a technical approach, while destructive thinking is a mental attitude. We create positive test cases to find bugs in software.*"

Brain Positioning

Remember that we damage the tester's attitude (and, thus, our bug finding abilities!) when we start proving that the software works, instead of proving that the software has bugs. We should always have prosecutor's stance: "Guilty!" towards software.

Build your tester's mentality on a lack of trust and destructive thinking, and you'll experience the magic: you'll start **feeling** software! Out of nowhere you'll be getting bug finding ideas targeting precisely those areas of software that contain bugs. I don't know HOW it works, but I know that it DOES work. **The right attitude invokes amazing forces that blow winds into the sails of whatever you do.**

Intro To Special Skills In Bug Finding

As you know, testers find bugs by executing test cases (there are, of course, other ways to test, like ad hoc testing, but the professional way is to use test cases).

We've already talked about the **formal** side of test cases.
Today we'll talk about test case **content**.

The art of creating test case content lies in finding those golden

- ✓ **ideas**
- ✓ **scenarios**
- ✓ **expected results**

that are well geared to assist us in finding bugs.

Test case content is created by using a set of special techniques (another term is *methods*) called *black box testing methodology*.

Below are the methods that we'll learn today:

1. Dirty List/White List
2. Test Tables
3. Flowcharts
4. Risk Analysis
5. Equivalent Classes
6. Boundary Values

Please note that there are also other methods used by black box testers. I picked these 6 because:

- They are very reliable.
- They are well fit for start-up environment.
- They have been successfully used by the PayPal QA Team.

Black Box Testing Techniques

DIRTY LIST/WHITE LIST

This method has two stages:

1. DIRTY LIST

This is very simple – type into a text file **ALL** ideas about testing that occur to you during (or after):

- Spec reading
- Exploration
- Getting feature info from other sources

Do not filter or analyze stuff; just allow your creativity and inspiration to flourish. The phrase "ALL ideas" has the widest meaning: test case ideas, ideas for scenarios, ideas about expected results, ideas for automation helpers, questions for PM and developer, WHATEVER. Don't think that some ideas might be ridiculous or impractical. This is a brainstorm where you allow your imagination to soar without any restraints.

The content of the text file* will be the dirty list.

**Some folks (like me) prefer to use paper for the dirty list/white list method. Whatever works!*

2. WHITE LIST

Here you start to **analyze** what you've come up with during your dirty list creation. Things that make sense are transferred into another text file called a *white list*. Your analysis will be based on a variety of things: the requirements from the spec, common sense, other test methods (e.g., Boundary Values), communication with the developer and the PM, etc. The bottom line is that you allow on the white list only those things that in your opinion will help in some way or another to find bugs.

BTW

I recommend that you divide your white list into two parts:

1. Items for test cases
2. To do list

You'll fill up the first part with things you will use in your test cases – e.g., ideas about the expected results.

You'll fill up the second part with testing-related action items – e.g., "Talk to Billy about a new DB table for this feature."

When you transfer items to your white list, you may want to:

- **Modify them**
- **Group them**
- **Add more detail to them**

Very often, the first white list turns into a second dirty list, so you create another white list based on this, and so on, until you get a finished product that can assist you to create great test cases and do other test preps (e.g., write an automation helper). It might be a little tricky to decide when your white list is finished – there is always room for improvement. My own approach is simple: I start working on test cases as soon as I feel that my white list gives me enough details to proceed with.

Let's start mastering the dirty list/white list technique right now. Take a sheet of paper and spend 15 minutes creating a dirty list of ideas on how to test a soda vending machine. For example:

- Does the buyer get the correct type of soda?
- What if the buyer presses the "Buy" button two times?
- Check to make sure the buyer gets correct change.
- Check to see whether foreign coins are accepted.

After your dirty list is done, spend another 15 minutes creating a white list. And then, spend 30 minutes creating test cases using this format:

Idea: _____

Steps: _____

- 1.
- 2.
- ...
- n.

Expected result: _____

This exercise is really useful. Whatever you do in the future, the dirty list/white list method will serve you well in **any kind** of project that requires planning!

TEST TABLES

Here is a mock-up of the first page of “Sign up” flow (*flow* means *scenario*):

The form consists of a yellow rectangular box with a thin black border. Inside, there is a text input field labeled "ZIP code*" with a placeholder. Below the input field is a blue "Continue" button. At the bottom left of the box, there is a small note: "*required".

Let's put possible conditions and inputs into the first table:

Table 1

ZIP code	zip_elem_8	zip_elem_7	zip_elem_6	zip_elem_5	zip_elem_4	zip_elem_3	zip_elem_2	zip_elem_1
ZIP code entered?								
Yes	x							
No		x						
Entered value								
5 digits		x						
4 digits			x					
6 digits				x				
has letter					x			
has special character						x		
has space							x	

As you can see, there are two subgroups:

ZIP code entered? is about condition.

Entered value is about data.

Each element (e.g., "Yes" for **ZIP code is entered?**) has its own unique ID (e.g., zip_elem_1). I chose "zip_elem" as a short form for "zip code table element".

Now let's combine the elements from Table 1 into finished scenarios and add some details to the new elements:

Table 2

ZIP code	zip_comb_1	zip_comb_2	zip_comb_3	zip_comb_4	zip_comb_5	zip_comb_6	zip_comb_7
Positive tests							
5 digits, 12345	x						
Negative tests							
Null input		x					
4 digits, 1234			x				
6 digits, 123456				x			
has letter, u2345					x		
has special character, 1234\$						x	
has space, 12 45							x

Table 2 is harder to create than Table 1, because we have to start figuring out how the elements can be combined. For example, zip_comb_1 = zip_elem_1 + zip_elem_3, but we cannot combine zip_elem_2 and zip_elem_3.

Here again, each element has a unique ID. Why do you need those IDs? Mostly because it's easier to refer to a particular element when it's properly indexed.

BTW

In this particular example, we played with:

- **Conditions** (ZIP code entered?) AND
- **Data** (entered value)

In other cases, you might have only data OR only conditions.

We have come up with 7 scenarios (zip_comb_1 to zip_comb_7 inclusively) that we can use to create test cases. In the test cases with scenarios from zip_comb_2 to zip_comb_7 inclusively, the expected result is the error message:

"Oops, error on page. ZIP code should have 5 digits"

In case of zip_comb_1, it's the next page:

The form is a registration page with the following fields:

- First Name* (highlighted in yellow)
- Last Name (highlighted in green)
- Email* (highlighted in blue)
- Password* (highlighted in red)
- Confirm Password* (highlighted in orange)
- Register button

A note at the bottom left says "*required".

Remember how we purchase books at ShareLane:

1. Register
2. Log in
3. Select book
4. Add book to shopping cart
5. Checkout

With Tables 1 and 2, we came up with scenarios for the first page ("ZIP code") of step 1: "Register". The second page ("First Name" and other fields) will have its own test Tables 1 and 2.

Each Web page for each stage of the book purchase process will have its own Table 1 and Table 2.

Most of the black box testing (for Web-based apps) involves an interaction with a sequence of Web pages that are integrated with each other and create certain combinations that lead us to expected results.

In some cases, we must have Table 3, Table 4, and so on reflect complex scenarios.

In some cases, we have to integrate the last table of one Web page with the first table of another Web page.

In some cases, Table 1 is enough to start creating test cases.

So, there is a huge amount of situations and options here. Be creative!!!

I know that once you start using test tables for more or less complex scenarios, you might feel lost. My only advice is: Just keep practicing.

When I took a Unix class some time ago, there was a long lecture dedicated to regular expressions (the art of finding patterns in text). After the lecture, my pal said that he felt like he had had brain surgery, and all of us completely shared his opinion. I'm serious: my brain was hurting. Don't be surprised if you feel the same trying to combine complex scenarios using test tables. But again, don't worry – the Test Tables technique takes time, but once you get it, you'll never forget it.

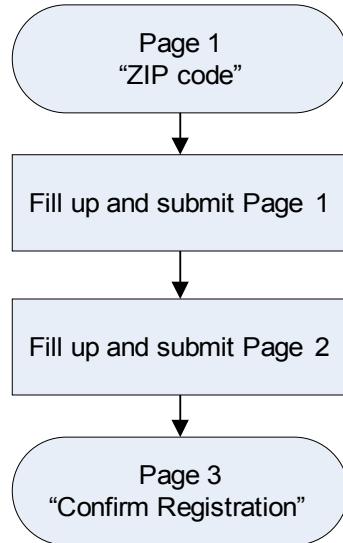
I don't want to overload you with work, but please do this exercise for practice: Create Table 1 and Table 2 for the flow "Login". Below are suggested subsections for Table 1:

Username entered?
Password entered?

Username valid?
Password valid?

FLOWCHARTS

When we talked about the SDLC, I mentioned process flowcharts as a means to improve specs. A **process flowchart is a graphic representation of a specific process**. Flowcharts allow for different levels of abstraction. For example, we can represent the process of registration in this form:



This flowchart and its sister from the lecture about the SDLC are:

- Similar, because they show the logic of registration
- Different, because they have a different number of details – or, in other words, a different level of abstraction

When you create your flowcharts, you'll use the level of abstraction required for your concrete task. For example, if we are testing registration, we'll need a more detailed flowchart of the registration process compared to a situation where we want to include a registration flowchart as a part of some other process – e.g., the process of making a book purchase.

BTW

The degree of abstraction is reflected in 2 terms: "high level" and "low level."

When you are asked, "Can you give me a high level overview?" it means that someone wants to have a general conceptual understanding about the subject.

When you are asked about a low level overview, you are being asked to provide small details.

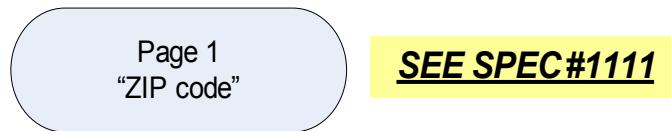
Here are basic building blocks (or "shapes") for flowcharts.

	This is the point of start/end of the flowchart. This block contains text with the name of the Web page, or "Start"/"End," or whatever text is needed.
---	--

	This block denotes some process – e.g., "Fill up and submit Page 1."
	This is decision point – i.e., depending on certain conditions, the process can take different passes from this point on. The text of the condition – e.g., "Data valid?" should always be specified in this block.

Below are several recommendations about flowcharting:

1. Before you start drawing a flowchart, give a name to the main process that is going to be described in the flowchart – e.g., "Registration".
2. Draw the main process first, and then expand it with details. In other words, start from the high level.
3. Put short yet informative text into the flowchart blocks.
4. Give references for useful info. For example, we can put the spec name to the left of starting block.



5. Make the process flow from top to bottom and from left to right.
6. Avoid the crossing of arrows.
7. Always double-check your flowchart to make sure that it correctly reflects the process. Use the spec or another source of knowledge.
8. Flowcharting can get really fancy: there are tons of different shapes that can be used. I personally use 3 basic shapes and have no problems with this. If you learn 15 shapes, it won't hurt as long as you accurately chart a process.

Let's do a quick training. Draw a flowchart for making tea. This task seems easy and fast to do at first glance, but check it out and you'll be surprised how many blocks you'll have for such a trivial process. Below are some ideas:

- Is there water in the teakettle?
- Is the stove on or off?
- Is the water in the teakettle boiling?

Use Microsoft Visio (you can download and install an evaluation copy) or just an old good sheet of paper.

Flowcharts are amazing visual resources for generating testing ideas. Besides, as with other testing methods, the process of the creation of the flowchart:

- Allows the tester to look at software from different angles

- Brings up a number of questions that didn't come up during plain spec reading

BTW

Truly amazing results can be achieved by *combining* testing techniques.

For example,

- You can create a dirty list and create Table 1 as a white list.
- You can make a dirty list by looking at Table 1 and then create Table 2 as a white list.
- You can create a flowchart before creating Table 2.
- You can explore LOTS of other possibilities.

The enormous number of possibilities gives you many opportunities to be creative. Nothing is set in stone in software testing. Use your imagination to turn whatever you've learned during this course into effective solutions applicable to concrete situations.

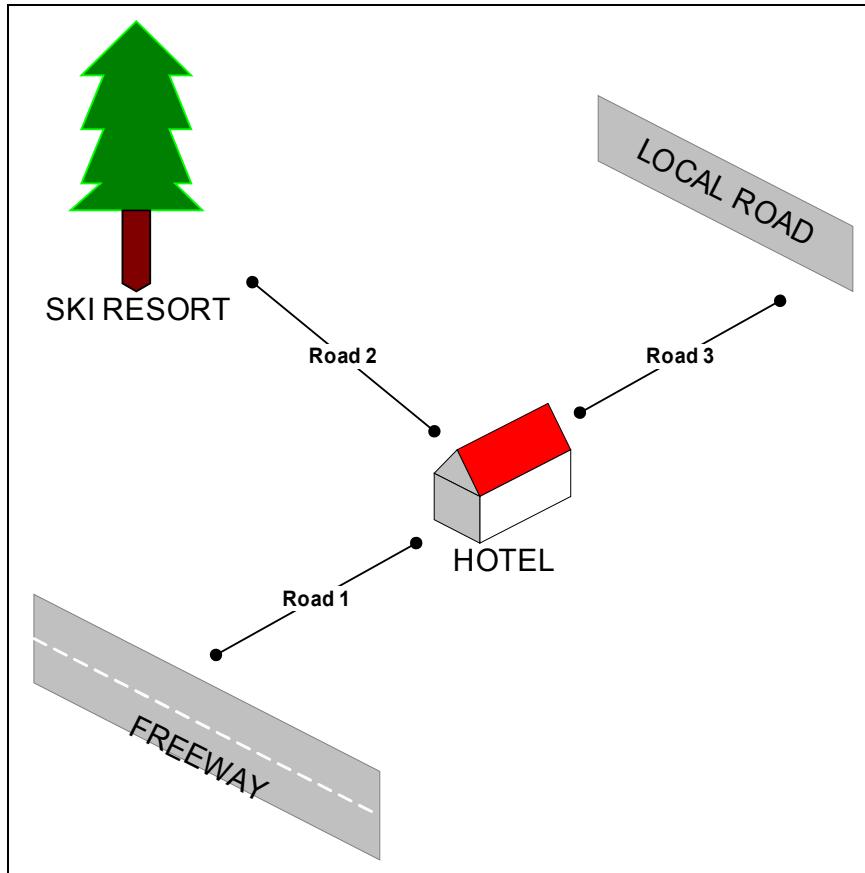
RISK ANALYSIS

Imagine that we've just bought a hotel in California somewhere in mountains of Sierra Nevada. We've got no experience managing hotels, but we are pretty confident that we can handle it. Not a big deal, right?

There are 3 roads that lead to the hotel:

- Road 1 connects the hotel and the freeway.
- Road 2 connects the hotel and the ski resort.
- Road 3 connects the hotel and the local road.

Roads 1, 2, and 3 have the same length, width, and altitude.



Ten guests have already arrived, and 30 are going to arrive today. It has been snowing like crazy all night, and Roads 1, 2, and 3 are not usable. There is only one snow cleaning truck at our disposal, and it takes half a day to clean one road. The question is: What road shall we start with?

At first glance it doesn't seem like a big puzzle to solve:

"It's **absolutely obvious** that Road 3 should not be cleaned first, because it's only used by our poker buddies. Screw them."

"It's **absolutely obvious** that Road 2 should not be cleaned first, because 10 (peeps who would want to go skiing) is less than 30 (peeps who will take Road 1 to reach the hotel)."

"Therefore, it's **absolutely obvious** that we should start cleaning up Road 1."

Does it sound logical? If "yes", note our confidence about how **absolutely obvious** this stuff is.

Here is my suggestion. Let's put the implementation of our genius plan on hold for a minute and ask James H. (a manager who has worked at the hotel for 20 years) for his opinion:

Situation Number One

Question: "James, what road shall we clean first?"

Answer: "You know, boss, it's very simple. Those 10 who have already arrived came here to throw snowballs, drink beer, and sleep. I know this, because those crazy bastards come here every year, and it's hard to imagine people who care less about skiing. So there is no need to clean up Road 2."

"I also know that 16 of the 30 arriving today are from a company that will depart this morning from Reno (I talked to one of them yesterday). They are going to take the road [James shows the road on the map] that merges with the local road connected to Road 3. So they'll take Road 3.

"Now, look at the reservations. Twelve of the 14 remaining guests live in San Francisco. I've just heard on the radio that the highway I-80 leading from San Francisco to us has terrible traffic. I'm going to assume that those 12 will leave after work, departing San Francisco at 5:00 p.m., and merge onto Road 1 no earlier than 9:00 pm.

"Therefore, I suggest that we clean Road 3 first and, after that, clean Road 1.

"BTW, 2 guys are coming from Arizona. I'll call them right away, explain the situation, and decide on the best route for them."

Situation Number Two

Question: "James, what road shall we clean first?"

Answer: "You know, boss, first we should start cleaning Road 2. All of our 10 guests are skiers. Besides, the 30 remaining people will most likely go to the ski resort first and come here only in the evening – I ordered 30 skiing passes starting with today, so they won't want to lose the whole day."

Situation Number Three

Question: "James, what road shall we clean first?"

Answer: "No problem, boss. We'll clean both Road 1 and Road 2. They have the same importance for our guests. I'll call my friend Steve who has his own snow-cleaning truck and lives next to Road 2, and he can help us out. His is a nice person, and he won't charge us much. He'll take care of Road 2, and we'll take care of Road 1. That way, we'll clean up both roads before noon."

The moral of the story is: **Conclusions based on subjective assumptions might lead us into situations where resources are wasted. At the same time, conclusions based on credible information will most likely lead us to effective solutions even if we don't have many resources.**

Technically speaking, our wise manager James did **risk analysis**. His risk analysis was accurate because he:

- Had correct information
- Knew how to use that information

Risk analysis is the evaluation of data or expectations with the purpose of setting priorities.

Here is the tester's approach towards risk analysis:

1. **Get information** about the subject of testing (e.g., the feature "Shopping cart").

The tester can get information from specs, statistics, live data about the Web site usage, communication, etc.

2. If possible, **get an opinion** from the person who knows the subject better than the tester does (e.g., the PM).

That person can be the PM, programmer, business analyst, somebody in accounting and finance, etc.

3. **Do risk analysis** (e.g., evaluate different patterns of usage in the shopping cart and determine the best ways to test it*).

* *Yep, ShareLane is a simple application, and there is not much to do here with Risk Analysis – just grasp the concept.*

Here is another example. Let's assume that we have to test the following functionalities (forget about ShareLane for a minute):

- a. Sales between users in America
- b. Sales between users in Japan
- c. Sales between users in America and Japan

Let's assume that we are selling repair parts for cars, and our Web site connects buyers and sellers.

Let's use Test Tables. First, let's create subgroups of the elements:

Table 1

Sales	sale_elem_1	sale_elem_2	sale_elem_3	sale_elem_4
Seller				
American	x			
Japanese		x		
Buyer				
American			x	
Japanese				x

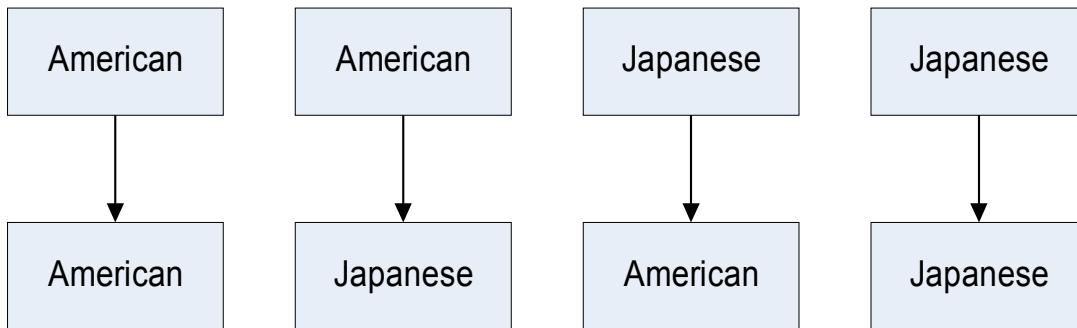
Second, let's combine those elements into scenarios.

Table 2

Sales	sale_comb_1	sale_comb_2	sale_comb_3	sale_comb_4
American Seller -> American Buyer	x			

American Seller -> Japanese Buyer		x	
Japanese Seller -> American Buyer			x
Japanese Seller -> Japanese Buyer			x

Very nice! We have 4 scenarios. Let's present them in the form of a flowchart:



So, we have 4 flows. Let's play our favorite game, "absolutely obvious":

1. The most popular transaction must be **Japanese Seller -> American Buyer**, because the American market has a huge number of Japanese cars.
2. Less popular is the flow **American Seller -> American Buyer**: Yes, there is a substantial demand for repair parts for American cars, but there are many other venues to get those parts besides our Web site.
3. Next we have the flow **American Seller -> Japanese Buyer**. We put it in 3rd place because it's less popular than items 1 and 2, but more popular than item 4.
4. The least popular transaction is **Japanese Seller -> Japanese Buyer**. Why? Because it's absolutely obvious that the Japanese have their own internal infrastructure of car parts sales, and nobody uses our site there.

It all seems pretty logical until we start digging.

Question: What is the source of our information – e.g., how do we know that our Web site is popular in the 1st scenario (**Japanese Seller -> American Buyer**)?

Answer: We've heard it somewhere or read it somewhere, or just have a gut feeling that it is the case.

Let me ask you this: **What if whatever we've heard or read was outdated, used in a certain context, or just plain wrong? And how credible is that "source of data" called a *gut feeling*?**

Answer: ...

Another question: What does this really mean: "The American market has a huge number of Japanese cars?" How huge? What is the number or percentage of Japanese cars on American roads?

Answer: ...

Our conclusions, based on the "absolutely obvious", have been a house of cards...again...

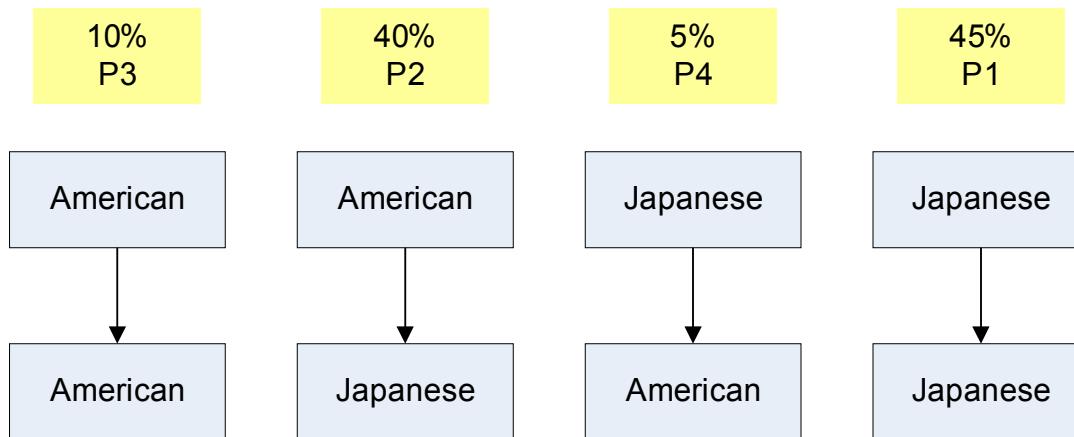
Here's the idea. Let's go to the PM, Amy, show her our flowchart, and ask for her opinion. Oops, it appears that Amy doesn't have live data about our Web site usage. But she tells us that the business analyst, Brian, must have the information we need. We take Amy with us and go see our man Brian.

"Hello, Brian. Can you help us? Here is a flowchart with 4 possible types of transactions based on the country of the seller and buyer. Can you give us an idea about the actual popularity of each of those types?"

Brian pulls up live data for the past two years, and off we go:

1. Most of the transactions took place between Japanese customers (**Japanese Seller -> Japanese Buyer**). Amy adds that, according to last marketing meeting, the Japanese market was declared to be the highest priority for the further growth of our company.
2. There are slightly less transactions for the **American Seller -> Japanese Buyer** segment. Amy adds that recently we closed two very important contracts with Ford and General Motors that makes us their single venue for Internet sales of repair parts for foreign customers.
3. There are much fewer numbers for **American Seller -> American Buyer** transactions. Amy comments that our competitors dominate here.
4. The least popular are the **Japanese Seller -> American Buyer** transactions. Amy says that Japanese car companies prefer to distribute repair parts directly to car dealers in America. That's why American buyers shop at cars dealerships instead of our Web site.

After that, we ask Brian to give us the concrete percentage for each transaction type (among the 4 types) based on the number of actual transaction that took place for the past 2 years. If we have that percentage, we can prioritize each type.



Now, instead of conclusions based on unreliable assumptions, we have actual data and an expert opinion based on reality and experience!

BTW

How can these priorities help us with our test cases? They give us guidance about:

- What we shall test more thoroughly – i.e., P1 flows must get more love than P2 flows

- What should be tested first – i.e., P1 test cases (in other words, test cases with P1 flows) must be placed before P2 test cases inside the test suite

Besides, if your testing is based on priorities set after performing risk analysis, you'll always have answers to the questions: "Why did you test that feature this way?" and "Why did you dedicate this much effort to testing that feature?"

Don't be shy about asking for help regarding live data, opinions, etc. It's normal, and people usually love to help you.

EQUIVALENT CLASSES

An equivalent class is a set of inputs that are treated the same way by the software under certain conditions. In other words, under certain conditions, the software must apply the same logic to each element of the equivalent class.

The ShareLane PM, Linda, wrote a spec about the feature that gives discounts to those customers who buy many books. Here is the table from her spec:

Number of books	Discount Rate, %
20-50	2
50-100	3
100-500	4
500-1000	5
1000-5000	6
5000-10000	7
10000 or more	8

We can immediately detect two spec bugs:

1. It's not clear what discount rate should be applied if the number of books is *exactly* 50, 100, 500, 1000, 5000, or 10000. For example, value "50" belongs to two baskets: one basket has a discount rate of 2%, and the other is 3%.
2. This table is not complete, because it's not specified what happens if we have less than 20 books. Logically, it must be 0 (zero), but it makes sense to have that basket in the spec anyway, just to make sure that we have a complete list of inputs.

We discovered those bugs during our spec review and sent an email to Linda requesting that she fix them. Here is how the table looks after the fixes:

Number of books	Discount Rate, %
1-19	0
20-49	2
50-99	3
100-499	4
500-999	5
1000-4999	6
5000-9999	7
10000 or more	8

In other words, we have 8 equivalent classes:

Class 1	1-19
Class 2	20-49
Class 3	50-99
Class 4	100-499
Class 5	500-999
Class 6	1000-4999
Class 7	5000-9999
Class 8	10000 or more

Each value inside a given class (e.g., Class 2) must be equally treated by ShareLane software. For example, if one user buys 21 books and another user buys 43 books, they must each get a 2% discount because those values belong to the same class (Class 2).

Each class has two parts:

1. **Range of inputs** – e.g., 20-49 inclusively for Class 2
2. **Logic for output** – e.g., Class 2 inputs are treated with a 2% discount

Brain Positioning

It's also very important to understand the critical role of **conditions**. In our case, everything is simple: we only care about how many books are purchased. But what if we were going to give discounts to users from the U.S. only? In that case, our set of 8 equivalent classes would make sense only under this condition: "Checkout is made by a user from the U.S." Other conditions would bring to life other sets of equivalent classes – e.g., the condition "Checkout is made by a user from outside the U.S. " would give birth to a set that consists of only 1 class:

Class 1 / 1 or more books / Discount is 0%

So what's so cool about equivalent classes? When we divide inputs into equivalent classes, we can:

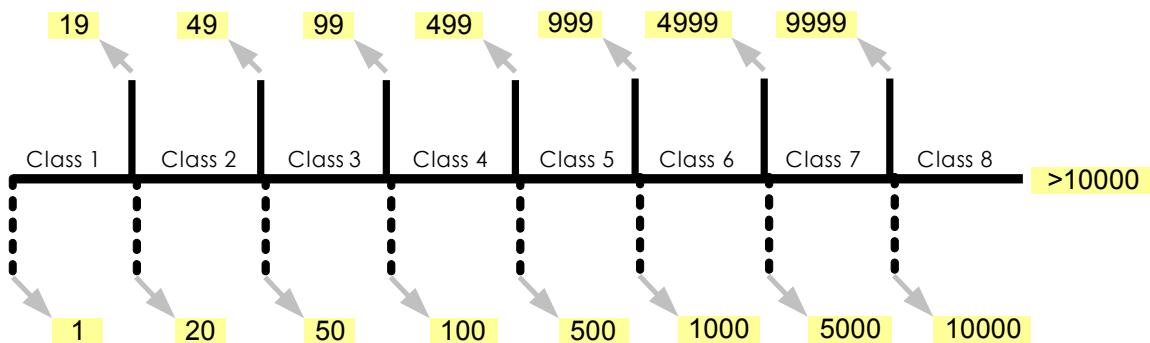
- Clearly see the connection between the ranges of inputs and the logic of the code
- Apply the technique called **Boundary Values** to select a small golden number of inputs out of the enormous number of possible inputs, and use that small number to hit the software in places where it's likely to break

BOUNDARY VALUES

Look at the picture below.

The **dashed** vertical line is the first value of the class (lower boundary).

The **solid** vertical line is the last value of the class (upper boundary).



For each equivalent class, we have only 1 out of 3 possibilities:

1. There is only a lower boundary (Class 8).
2. There are both lower and upper boundaries (Classes 1-7 inclusively).
3. There is only an upper boundary (we would have that class if we had a class unrestricted by a lower boundary – e.g., "any value below 1").

The Boundary Values technique relies on the idea that **software tends to break at the upper and lower boundaries of equivalent classes**.

BTW

Term *boundary testing* refers to usage of Boundary Values technique

Here is the full version of Boundary Values technique for one equivalent class.

- A. First, we test the **lower boundary of the class** (if the lower boundary exists).
- B. Second, we test the **upper boundary of the class** (if the upper boundary exists).
- C. Third, we test **any value inside the class** (if the class consists of 3 or more values).
- D. Fourth, we test the **upper boundary of the class that precedes the tested class** (if the preceding class exists).
- E. Finally, we test the **lower boundary of the class that follows the tested class** (if the following class exists)

Here's the same thing in the form of a table (see next page):

A-to-E table

Test ID	Boundary	Class	Do this test if...
A	Lower	current	lower boundary exists
B	Upper	current	upper boundary exists
C	<Between>	current	class consists of 3 or more values
D	Upper	preceding	preceding class exists
E	Lower	following	following class exists

A, B, and C are positive tests. D and E are negative tests.

Let's do boundary testing and see each of the 5 elements (A-E) in action!

SL

1. Go to Test Portal>Application>Source Code>shopping_cart.py. You can find the list of bugs below the “View bugs” subsection.
2. Open another Web browser window, create an account on ShareLane, login, add a book to the shopping cart, and click the “Shopping Cart” link.

3. Start learning the material from the table below (remember to click button "Update" after you enter new value into the text box "quantity")

		Expected code with singled out part being checked
Test ID and description		Possible issue with code exposed by test and example. <u>BUG #</u> in shopping_cart.py
Number of books inside the shopping card		Expected discount %
A. Test lower boundary of the class: 50	if $q \geq 50$ and $q \leq 99$: discount = 3	No equal sign and/or error in value of lower boundary. <i>Example:</i> $if q > 50 \text{ and } q \leq 99:$ discount = 3 <u>BUG #2</u> 3%
B. Test upper boundary of the class: 499	if $q \geq 100$ and $q \leq 499$: discount = 4	No equal sign and/or error in value of upper boundary . <i>Example:</i> $if q \geq 100 \text{ and } q \leq 399:$ discount = 4 <u>BUG #3</u> 4%
C. Test any value inside the class 550	if $q \geq 500$ and $q \leq 999$: discount = 5	Error in more than (>) and less than (<) signs. Example: $if q \leq 500 \text{ and } q \geq 999:$

	discount = 5
	<u>BUG #4</u>
	5%
D. Test upper boundary of the class that precedes tested class 19	<p>if $q \geq 20$ and $q \leq 49$: $discount = 2$</p> <p>Here we check two things:</p> <ol style="list-style-type: none"> Logical jump from upper boundary of preceding class to lower boundary of given class <p><i>Example:</i></p> <p><i>if $q \geq 12$ and $q \leq 49$: $discount = 2$</i></p> <p>The programmer used 12 instead of 20, so the logic towards some members of preceding class (values 12-19 inclusively of Class 1) is the same logic towards members of the given class (Class 2). The jump should have taken place between 19 (0% discount) and 20 (2% discount), but it didn't happen.</p> <p><u>BUG #1</u></p> <ol style="list-style-type: none"> Logical "AND" versus logical "OR" <p><i>Example:</i></p> <p><i>if $q \geq 20$ or $q \leq 49$: $discount = 2$</i></p> <p>In case the programmer mixes up AND and OR, the code matches any value. Read more about this issue in Test E.</p>
	0%
E. Test lower boundary of the class that follows tested class: 10000	<p>if $q \geq 5000$ and $q \leq 9999$: $discount = 7$</p> <p>Here we check two things:</p> <ol style="list-style-type: none"> The logical jump from the upper boundary of the given class to the lower boundary of the following class <p><i>Example:</i></p>

```
if q >= 5000 and q <= 10000:  
    discount = 7
```

2. Logical "AND" versus logical "OR"

Example:

```
if q >= 5000 or q <= 9999:  
    discount = 7
```

Here is how the code works: if we have a block of conditional logic – e.g.,

```
if <condition1>:  
    <do that>  
elif <condition2>:  
    <do that>  
elif <condition3>:  
    do that
```

then the value fed to this block is being checked against each condition from the top to bottom until there is a match between the condition and the value (e.g., condition: *if a = 5, and a = 5*); if no match happens for all conditions, program execution goes to the next line of code after the conditional block. If there is a match, then *<do that part>* part is being executed. The important thing is that if a match happens, the program doesn't execute any further conditions in this block.

For example, this Python code:

```
a = 2  
if a == 2:  
    print "foo"  
elif a == 2:  
    print "bar"
```

prints "foo"

Now go to **Bug #5**

Here is what happened in shopping_cart.py. There is a set of values without a match before the condition:

	<p><i>if q >= 5000 or q <= 9999:</i></p> <p>One member of this set is 10000. So, after we have a match between 10000 and that condition, the discount value is set to 7 and the program exits execution of this conditional block. That's why values ≥ 10000 will always get discount 7 in spite of the fact that the condition that is intended for values ≥ 1000 has the correct code:</p> <p><i>if q >= 10000: discount = 8</i></p> <p>In other words, the two lines of code above will never get executed.</p> <p>BTW, the words in bold below are terms from computer science: the value is tested against conditions; the result of each test is either TRUE or FALSE.</p> <p>When we talked about the match between value and condition, we implied that the result of the test was TRUE.</p>
	8%

Below is a set of inputs for performing boundary testing on all equivalent classes for discount functionality:

Class	Input	Discount %
1	1	0
	5	
	19	
2	20	2
	27	
	49	
3	50	3
	74	
	99	
4	100	4
	129	
	499	
5	500	5
	612	

	999	
6	1000	6
	2743	
	4999	
7	5000	7
	8765	
	9999	
8	10000	8
	17654	

We have 23 test cases to cover 7 closed (each of Classes 1-7 inclusively has both lower and upper boundary) and 1 open (Class 8 has no upper limit) equivalent classes. There are two very cool facts here:

1. We covered an unlimited number of inputs with 23 test cases.
2. We used a methodology that hits the software right at its breaking points!

Now, what if we wanted to test Class 2 in isolation from its sisters? We would need 5 test cases:

Boundary check	Input	Discount %	Type of test – Positive/Negative
D	19	0	N
A	20	2	P
C	27	2	P
B	49	2	P
E	50	3	N

We've used the most advanced examples to learn Boundary Values technique. Now, let's see how simple this can really be.

Example

As you recall, a ZIP code must consist of 5 digits. Let's apply the "A to E" thing:

- A. 5
- B. 5
- C. 5 (no need for this test because this class has less than 3 values)
- D. 4
- E. 6

Therefore, we have only 3 test cases (input "5" is positive testing; inputs "4" and "6" are negative testing), because the equivalent class is closed and consists of only 1 member (5).

BTW**SL**

Go ahead and test ZIP code field with this set: 4,5,6. File a bug using Training BTS if you find a problem (fill up only Summary).

When You Start To Implement Testing Techniques...

Here is a very important point to conclude our talk: **the tester should be a master of testing techniques, not their slave.** When you start to implement the black box testing methodology that you've learned, you'll probably feel frustrated, and in some cases you might start to think more about the implementation of methods than about finding bugs using those methods. To avoid that, you need a lot of practice to get up to speed with each method; in other words, you must become an expert in each of them. Remember our example with chess – **the rules are simple, but walking the walk is not.** So, PRACTICE A LOT! You have ShareLane and Test Portal for hands-on experience, and you have all the necessary theoretical knowledge to start practicing and become experts in the black box testing methodology.

Lecture Recap

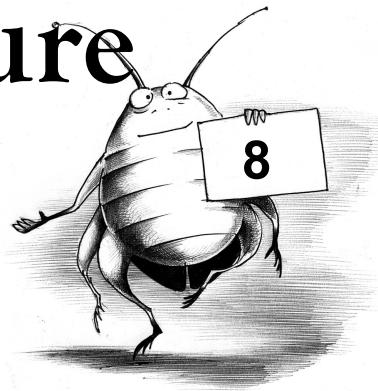
1. Software is a shelter for bugs.
2. A lack of trust and a destructive attitude towards software are the foundation of an effective tester's mentality.
3. A dirty list is about creativity and inspiration. A white list is about analysis.
4. Test Tables are an excellent way to present conditions and/or inputs in a structured way and combine them into scenarios.
5. A flowchart helps to present a process (e.g., a test scenario) in a visual way.
6. Amazing results are produced by combining different methods.
7. Risk analysis is the evaluation of data or expectations with the purpose of setting priorities.
8. An equivalent class is a set of inputs that are treated by the software the same way under certain conditions. In other words, under certain conditions, the software must apply the same logic to each element of the equivalent class. An equivalent class can consist of only one member.
9. Boundary testing checks certain points inside of equivalent classes for the purpose of detecting buggy parts of the code.

10. Testing techniques make up your toolbox. Use whatever you need from that box to find the best test scenarios. Creativity and practice are the keys here.

Questions & Exercises

1. How does a tester define software?
2. Which attitudes make up the foundation of a tester's mentality towards software?
3. List the black box testing methods you've learned today.
4. What is gist of the dirty list/white list method?
5. How many tables are needed to generate scenarios using Test Tables technique?
6. What is the main advantage of flowcharts for testers?
7. Who can assist the tester to analyze risks?
8. What value is there in assigning priorities to features?
9. Describe each step ("A to E") of the Boundary Values technique.
10. How can combining methods bring more effectiveness into our testing?

Lecture



Bug Tracking

Lecture 8. Bug Tracking.....	212
Quick Intro.....	213
The Bug Tracking System.....	213
Bug Tracking Procedure.....	251
Quick Closing Note About BTS And BTP.....	255
Lecture Recap.....	255
Questions & Exercises.....	257

The truth is incontrovertible.
Malice may attack it.
Ignorance may deride it.
But in the end, there it is.
- Winston Churchill

Sometimes you have to write down unwritten rules.
- Daniel Kionka

Quick Intro

As you know, the purpose of testing is to find and address bugs before they are found and addressed by our users. So far our focus has been on bug finding, and now it's time to talk about addressing bugs. Let's review the main concepts about addressing bugs:

After a bug is found, a tester must:

- File the bug into the bug tracking system

After the bug is fixed, a tester must

- Verify that the bug was really fixed
- Check to see if new bugs have been introduced during bug fixing.

BTW, remember that a. and b. are called regression testing.

The process that begins with filing a bug into the **bug tracking system** is called the **bug tracking process**. Every time you file a bug into the bug tracking system, a new instance of bug tracking process is automatically launched.

Here is the plan for this lecture:

First, we'll talk about the bug tracking system (let's shorten it to the BTS).

Second, we'll talk about the bug tracking procedure (let's shorten this to BTP), i.e., the set of rules about how the bug tracking process must function.

The Bug Tracking System

Let's forget about software for a minute. Let's say that you are test-driving a car. You check out how fast the car gains speed, how it handles turns, if controls like the audio volume are reachable, etc. After the test drive is finished, you write down on a notepad whatever didn't meet your expectations.

Example

Line 1: "navigation system sucks"

Line 2: "acceleration delay after gas pedal is pressed"
Line 3: "trunk is too small"

That notepad is the simplest version of the BTS.

Here is a definition:

The BTS is the infrastructure that enables to

- **create,**
- **store,**
- **view and**
- **modify**

information about bugs.

As a rule, software companies use professional BTS software. Some BTS software is free, like the most popular BTS, Bugzilla; some can cost tens of thousands of dollars (for example, Test Director by HP). I have used many of them, and although I personally prefer Test Director, I must say that **Bugzilla is an ideal bug tracking solution for most software start-ups.**

BTW

In this course, we will consider a BTS structure and a BTP that are not directly linked to any existing BTS software. Why? Because I want to give you a universal knowledge that will allow you to easily deal with any concrete BTS and BTP.

BTW

From now on, depending on the context, the term "bug" will mean:

- a mismatch between the actual and the expected (our classic definition)
and/or
- a BTS record of that mismatch

Example

- "I found a bug in Checkout."
- "There are 25 open bugs in Bugzilla."

As we know, **the fact that a bug has been discovered has no practical value until we share information about that bug.** How can we share the information? We can make a phone call, send an instant message or email, talk personally, etc. The standard way to report bugs in software companies is to file a bug into the BTS. So, another important function of the BTS is the **communication medium** between SDLC participants. Of course, bugs are communicated in other ways, too (e.g., we can talk about bugs at the Go/No-Go meeting), but the rule remains: **Each and every bug found must be filed into the BTS.**

Question: How can we get information about the bug itself and the activities surrounding it?

Answer: We must check the **bug Attributes** in the BTS.

Example

One of the bug Attributes is **Status**. Status has 3 possible values:

- Open**
- Closed**
- Re-open**

At any given moment, a concrete bug can have only 1 of those 3 values.

Once a new bug is filed into the BTS, its status is automatically set to "Open". After the bug is fixed and successfully regressed, the tester changes its status to "Closed"; if the same bug appears again, the tester reopens the bug by changing the bug status to "Re-open".

BTW, transitions from Open to Closed and from Closed to Re-open are made according to the rules set in the BTP.

SL Now, do this quick exercise:

1. Create an account on ShareLane and login.
2. Go to Test Portal>Bug Tracking>Training BTS>Submit new bug.
3. File a bug about anything: e.g., "Today the weather is bad" (fill up only Summary).
4. View that bug and change its status to Closed.
5. View that bug and change its status to Re-open.

Bug Attributes are needed to:

1. Describe the problem with the software/spec
2. Reflect activities made towards solving the problem
3. Provide other useful info about the bug

Let's learn about those Attributes! I urge you to actively use Test Portal to file/modify your own bugs during this lecture.

Here is the list:

- ID
- Summary
- Description
- Attachment
- Submitted by
- Date
- Assigned to

- Assigned by
- Verifier
- Component
- Found on
- Version
- Build
- DB
- Comments
- Severity
- Priority
- Also notify
- Change history
- Type
- Status
- Resolution

ID

The bug ID is the unique identifier of each bug in the BTS. The BTS automatically assigns next available ID after a new bug is filed. The bug ID is not changeable. Among other usages, the bug ID is the universal way to refer to a concrete bug: "Hey Billy, how is the situation with 998?"

SUMMARY

The Summary is a short, informative message about the gist of the bug. As a rule, the text field for the Summary doesn't exceed 80 characters with spaces. The way a tester writes a Summary is a very good indication of his or her professionalism.

Example of a good Summary:

"Spec2345: 2% discount is given for 12-19 books inclusively"

Example of a bad Summary:

"problem with discount"

BTW



I recommend that you look into the official ShareLane BTS called the Bug Vault (Test Portal>Bug Tracking>Bug Vault) for examples of Summaries. **Click the bug ID to open a Web page with full details about that particular bug.**

The purpose of the Summary is to give the BTS user a **quick, powerful description** of the bug. A bug Summary is to a bug as a title is to an article. Professional journalists are specifically trained in the art of creating good titles. I recommend that you start practicing right now. It's not a big deal, really. Just start writing short Summaries of all the bugs you see in real life;

"I wanted a beer, but my wife bought me an ice cream."

"I wanted an ice cream, but my husband bought me a beer."

(BTW, bad Summary for 2 cases above would be, "I didn't get what I wanted.")

Here are some guidelines about Summaries:

1. Depending on the situation, start a Summary with one of the following:

- **The spec ID;** e.g., Spec2345
- **The name of the feature (or component);** e.g., Registration
- **The name of the file with bug;** e.g., shopping_cart.py

There are at least two reasons to do this:

a. The ability to unite related bugs by the same first word(s) in summaries. Bugs can be related because they were found while testing the same spec (e.g., Spec #1111 "New User Registration") or the same functional area (e.g., "Checkout"), etc.

- On the one hand, the benefit here is that the BTS user can sort a list of bugs by the first word in the Summary and thus see all related records in one group.
- On the other hand, if certain keyword (e.g., spec ID) is presented in the Summary, then the BTS functionality "Search" can be used to find and list in one group all the bugs with that keyword in the Summary.

BTW

It's a good idea to create a convention (inside your company) or at least some guidelines about how to start bug Summaries. For example, we can agree that if we begin a Summary with the spec ID, we should do it a **certain way**:

"Spec2345:" (no space)
"Spec 2345:" (space)
"#2345:" (pound, no word "Spec"), etc.

b. The ability to provide instant info regarding a buggy area. For example, if I put "shopping_cart.py:" as at the beginning of the Summary, then Billy, who is responsible for the shopping cart, will immediately know where to start looking for the problem.

2. While creating a Summary, think about your target audience – i.e., PMs, programmers, and testers. For example, if a certain component has an internal nickname like "cc" for "credit cards" you shall use it *only* if you are 100% sure that people from your target audience will instantly understand it. Always remember that bug filing is the process of sharing of information, so be clear. Concerning the level of abstraction, I prefer to pack as many details as possible into the Summary.

3. Sometimes it's really hard to create a good Summary. In this case, end the Summary with the phrase "**(see description)**" or "**(see attachment)**" – whichever is applicable. We'll talk about bug descriptions and attachments in a minute. An example of a Summary might be: "Homepage: wrong layout of the logo (see attachment)."

DESCRIPTION

The Description is a text area where the full details about the bug must be given.

SL I suggest the following format for bug Descriptions (check out the Description of Bug #1 in the Bug Vault):

Description: <provide information about what happens>

Steps to reproduce: <provide steps to reproduce the bug>

Bug: <state what's wrong>

Expected: <state what's expected>

Always remember that nothing is set in stone. In some cases, the bug Description can be just simple repetition of the bug Summary, or just the phrase, "see attachment" (when the attachment is self-explanatory); in other cases, you'll really need to make an effort and provide great detail.

An important thing to remember about **steps to reproduce**: some things, like professional slang or abbreviations adopted in the QA Department and widely used in test cases, may be absolutely incomprehensible to the PMs and programmers; always remember the purpose of the Description is to **share your knowledge about the bug in a way that is understandable to others**. Yes, sometimes it can be boring, and good Descriptions do take time and effort, but **bug reporting is the main way to address bugs, so each tester must make sure to provide crystal clear info about bugs**.

BTW

A very good approach is to copy and paste the interpreter error or log file extract or any other info that might be useful for debugging* into the Description.

**Debugging is the activity a programmer engages in when he or she is trying to identify a buggy piece of code and fix it.*

SL For example, you can copy interpreter error from here: Test Portal>More Stuff>Python Errors>register_with_error.py, and put it under the **Bug:** section of the Description.

SL Please, go ahead and file a bug against register_with_error.py using interpreter error in your Description.

Below is a table describing the most common Web page elements. This table will:

- Help you provide better Summaries, Descriptions, and test case steps
- Deepen your technical knowledge
- Give you an idea of the typical problems with each element

SL You can play with these elements here: Test Portal>More Stuff>Basic Web page elements.

Name of the

element	Example of the element
	Example of HTML code of the element
Description of the element	
Text	Price: \$10.00
	<p>Price: \$10.00</p>
Text is the most common element of the Web page. Problems with text include:	
<ol style="list-style-type: none"> 1. Wrong text (e.g., wrong text of the error message)* 2. No text (e.g., the code doesn't display the title of the book)* 3. Wrong text property: font, color, size* 	
Of course, the problems of "wrong content", "absence", or "wrong property" can happen with other elements too, e.g.:	
<ol style="list-style-type: none"> 1. Wrong image of the book 2. No image at all 3. Image with wrong size parameters. 	
Link	<u>Sign up</u>
	Sign up
Link (or hyperlink) is a navigation element. Link can point to	
<ul style="list-style-type: none"> - URL - Position on the Web page. Read on for examples. 	
Here are typical targets of links:	
<ol style="list-style-type: none"> 1. File, e.g., HTML file (Web page), PDF file, TXT file, Python file, etc. 2. Anchor on the same Web page 3. Anchor on a different Web page 4. “mailto” value 	
So, when a user clicks a link:	
<ol style="list-style-type: none"> 1. A certain file is opened (e.g., HTML, PDF, TXT files) or launched (e.g., the Python file shopping_cart.py dynamically generates the Web page "Shopping Cart"). 2. The Web browser scrolls down to a certain anchor on the same Web page. 3. The Web browser opens a new Web page and scrolls down to a certain anchor. 	

4. The Web browser invokes a new message instance of the default email client.

Examples

1. Click the link **Link to another page** on Test Portal>More Stuff>Basic Web page elements.
2. Click the link **Bug #3** under the section "View bugs" on the Test Portal>Application>Source Code>shopping_cart.py.
3. Click the link **Link to another page with anchor** on this page: Test Portal>More Stuff>Basic Web page elements.
4. Click the link **Mailto link** on the Test Portal>More Stuff>Basic Web page elements.

Below are some typical problems with links:

A. Misleading link: The link points to the wrong file and/or anchor, or mailto value.

B. Broken link: This term is usually used for problems with linked files. Here we have 2 variants:

- Web server responded that linked file doesn't exist at specified path. In that case we get the "**404 - File Not Found**" error.

- DNS cannot find IP address associated with that domain name. In this case we get "**DNS error - Cannot Find Server**".

BTW, the DNS (Domain Name System) is the translator from a domain name to the IP address of a physical computer on the Internet.

C. Misspelled mailto value: e.g., 'mailto:contactsharelane.com' instead of 'mailto:contact@sharelane.com' (note missed "@" sign).

See examples on Test Portal>More Stuff>Basic Web page elements inside the table, "Bad Links".

Check out the Downloads section of qatutor.com for automation tools that check for broken links.

BTW, linking the URL to a file can be presented in two ways:

- Relative URL: i.e., we assign a path to the file related to the directory where we are right now.

For example, if we are on the page http://main.sharelane.com/cgi-bin/main.py it means that we are inside the Web server directory of sharelane.com called 'cgi-bin'. The directory cgi-bin contains the file register.py, so we can put a link like that to register.py: Sign up on the HTML page generated by http://main.sharelane.com/cgi-bin/main.py.

- Absolute URL: e.g., we give the full path.

For example, if we are on the page http://www.google.com and want to reach

register.py on sharelane.com, we should do this: Sign up

Image



```
<img src='../images/product_2_large.jpg'>
```

Images are graphic files (as a rule, in a GIF, JPG or PNG format) that are referenced from the HTML code of Web pages. When a Web browser loads HTML code and "sees" a special tag: , it tries to download the referred image to the hard disk of the user. If the file exists, the user can see the picture. If the image is corrupted or the path to the image is wrong, then we have a situation called "**broken image**" and we can see only an image placeholder on the Web page that looks

like this:  (IE) or like this:  (Firefox).

Linked image



```
<a href='./show_book.py?book_id=8'>
<img src='../images/product_8_small.jpg'
height='150' border=0>
</a>
```

This is a link represented by an image instead of text. The linked image can have maladies of both images and links.

In the example of HTML code above, bolded is the code for a link and not bolded is the code for an image. If we put text The Moon and Sixpence instead of

`src='..../images/product_8_small.jpg' height='150' border=0>`
then we would get a standard textual link.

Text box	<input type="text" value="expectations"/> <input type="button" value="Search"/>
	<code><input type='text' name='keyword' size='50'></code>

The text box is a **one-line** field for text input. The text box (as well as other elements that we'll discuss further) serves as an input mechanism for a **Web form**. A Web form provides a way to submit one or a set of key-value pairs to a concrete file within the application core. Here is an example of HTML code for a Web form with a text box and a submit button (we'll learn about this button in a minute). The opening and closing tags for the form are bolded. Underscored is the code for the button.

```
<form action='./search.py'>

    <input type='text' name='keyword' size='50'>
    <input type='submit' value='Search'>

</form>
```

Here is what happens: after a user types *expectations* into the text box `keyword` and presses the "Search" button, key-value pair "keyword=expectations" is sent via the Web server to be processed by the application core file `search.py`.

BTW

- If you submit that form by using the Search functionality on `main.sharelane.com` **or**
- If you just type in this URL: `http://main.sharelane.com/cgi-bin/search.py?keyword=expectations`

You'll get the same Web page as a result.

Let's get back to the text boxes. Besides standard things like the ZIP code or email fields, text boxes are also used to enter the value for **captcha**. Captcha is an image with some alpha, numeric, or alphanumeric characters on it. Captchas are used to prevent bots (automated scripts) from submitting Web forms: the premise is that humans can distinguish text on an image, while bots cannot (some of them already can, BTW). Here is an example of a captcha from the Yahoo site. This captcha is displayed to prevent automatic registration of email accounts:



Regarding bugs connected with text boxes, we usually have

1. The wrong value for the text box attribute name. For example, if a programmer types `name='keywor'` (note: no "d" letter) instead of `name='keyword'` then no search would take place, because `search.py` expects the variable called "keyword". Go to this URL to check it out:
<http://main.sharelane.com/cgi-bin/search.py?keywor=expectations>

2. The wrong value for the attribute `maxlength` or no such attribute at all. `maxlength` is needed to limit the number of characters (including spaces) that can be typed into the text box. For example, Willy wanted to type this to limit search keywords to 25 characters:

```
<input type='text' name='keyword' maxlength=25>
```

but erroneously he typed:

```
<input type='text' name='keyword' maxlength=5>
```

As a result, a user can enter only a keyword with equal or less than 5 characters:

A screenshot of a search interface. There is a text input field containing the text "expec" and a blue rectangular button labeled "Search" to its right.

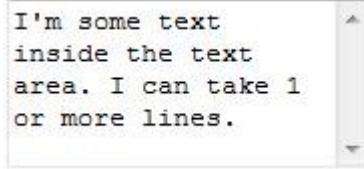
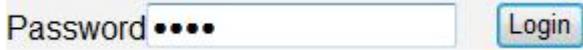
The first problem is discovered during functional testing; the second problem is discovered during UI testing.

BTW, very often a bug occurs because the DB column is able to store fewer characters than the user is able to submit. The grey box approach can help us find that bug. Do this:

1. Go to Test Portal>DB>Schema
2. Scroll down to table below this line: "mysql>describe users;"
3. Note the value of Type for Field `first_name`. It's **varchar(50)**.

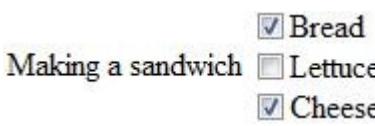
`Varchar(50)` means that the column `first_name` can store up to 50 characters, including spaces. So, if user enters 51 characters or more as his/her First name, 51st and following characters will be **truncated** (i.e., cut off), because the `first_name` column has a limit of 50 characters (for security purposes this exercise cannot be done on ShareLane).

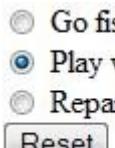
The easiest fix for this problem is to introduce a `maxlength` parameter to limit the

number of characters that a user can type into the text box.	
Text area	 <pre>I'm some text inside the text area. I can take 1 or more lines.</pre>
	<pre><textarea columns=10 rows=5 name='message_body'> I'm some text inside test area. I can take 1 or more lines </textarea></pre>
	<p>The text area is a multiline field for text input. An example of usage is the body of a message on an Internet forum. Usual text area problems include the wrong value for an attribute name or the wrong values for the columns and rows attributes.</p>
Password input box	 <p>Password <input type="password" value="*****"/> <input type="button" value="Login"/></p> <pre><input type='password' name='password'></pre>
	<p>The password field is a special one-line text box where the input is masked by asterisks or dots as the user types text. Character masking is required for security to prevent some friendly fellow from seeing your password and cleaning out your bank account.</p>
	<p>The usual problems with password fields are just like those with text boxes. In some cases, the programmer makes a mistake and uses a text box instead of the password field to submit a password. See Bug #1 in the Bug Vault.</p>
	<p>Below are two additional things that I want to mention here:</p>
	<p>Professional ethics: When somebody types a password in your presence you should <i>explicitly</i> turn your head (or your whole body) away from the person. It's extremely unprofessional to stare at the monitor/keyboard while somebody types a password. Your colleagues will surely appreciate the fact that you respect their privacy.</p>
	<p>Your privacy: Many software companies install software to capture keystrokes and screenshots on their employees' computers. This means that whatever you type or see on your monitor can be viewed by your employer. Just FYI...</p>
Drop-down	

menu	<p>Card Type:</p>  <pre><select name='card_type_id'> <option value='1'>Visa</option> <option value='1'>MasterCard</option> <option value='1'>AmEx</option> </select></pre>
	<p>A drop-down menu is a list of values to choose from. BTW, there are 2 bugs in the code above! Here is what happened: Each credit card has an internal index at Sharelane:</p> <ul style="list-style-type: none"> 1 for Visa 2 for MasterCard 3 for AmEx <p>(See Test Portal>DB>Data>cc_types)</p> <p>The developer made 2 mistakes. He put 1 for both MasterCard and AmEx. When we click "Make payment" button to complete the checkout, the content of the Web form is sent to checkout.py for processing. Part of the content is the key-value pair <code>card_type_id=<internal card index, e.g., 1 for Visa></code>. Here is how these 2 bugs echo in other parts of the system:</p> <ul style="list-style-type: none"> - The first digit of the value in the column <code>result</code> of table cc_transactions will always be equal 1 (Test Portal>DB>Data>cc_transactions) and - The second value on each line in the log file cc_transaction_log will always be equal to "V" (Test Portal>Logs>cc_transaction_log) <p>In this case, I would file 2 bugs with summaries like these:</p> <p><i>"checkout.py: value 1 is given for MasterCard in drop down menu card_type_id"</i> <i>"checkout.py: value 1 is given for Amex in drop down menu card_type_id"</i></p> <p>Why don't I recommend doing something like "<i>checkout.py: value 1 is given for all cards in drop down menu card_type_id</i>" in our case? Because the MasterCard bug is not related to the AmEx bug – <i>if you fix the problem with MasterCard, the bug with AmEx will not be fixed</i>. There is more to this situation, and we'll talk about it in several minutes.</p> <p>I recommend that you go to the Training BTS right now (Test Portal>Bug Tracking>Training BTS) and file those 2 bugs yourself (fill in only the Summaries and Descriptions).</p>

Back to the drop-down menus.... Now we know that the wrong value of the value attribute can give birth to a bug. In addition, we can also have problems with the wrong value of the name attribute, or with the completeness of the text parameters (e.g., if we accepted "Discover" then it should have been on the list, too) or their correctness (e.g., "Viva" instead of "Visa").

Radio button	 <pre data-bbox="448 623 1346 834"><input type="radio" name="morning" value="fishing"> Go fishing <input type="radio" name="morning" value="kids" checked> Play with kids <input type="radio" name="morning" value="dishwasher"> Repair dishwasher</pre>
	<p>The radio button is an element of a Web form that allows the selection of one, and only one, value from the group with the same name (within the same Web form). <i>The group must have 2 or more values. In the code above, the group's name is "morning".</i> In other words, logically the elements of the same group are mutually exclusive; e.g., in the above example you can select EITHER Go fishing OR Play with kids OR Repair dishwasher.</p>
<p>Typical problems with radio buttons include:</p> <ul style="list-style-type: none"> - Incorrect value for name - Parameter checked specified for a wrong value - Incorrect label ("Go hiking" instead of "Go fishing") <p>The radio button took its name from those old huge radio receivers with the big buttons: each button was associated with one frequency, so you could only listen to one radio station at any given moment.</p> <p>It's bad programming style to use a radio button if a group consists of only 1 element. In that case, if you select the button, you cannot clear it.</p>	 <pre data-bbox="448 1795 1158 1938"><input type="checkbox" name="option1" value="bread" checked>Bread <input type="checkbox" name="option2" value="lettuce">Lettuce <input type="checkbox" name="option3" value="cheese" checked>Cheese</pre>

	<pre><input type="checkbox" name="option3" value="cheese" checked>Cheese</pre>
	<p>Checkboxes are used for situations where there is a need to select 1 or more options. There are no relationships between text boxes, so in the above example you can check/clear any checkbox. In some cases, programmers add a JavaScript that can introduce logic when the checkboxes are dependent on each other (e.g., if I check Lettuce, Cheese might be automatically cleared).</p> <p>Checkboxes have the same maladies as radio buttons.</p>
Submit button	 <pre><input type='submit' value='Search'></pre>
	<p>The Submit button is needed to submit Web form content to the Web server. <i>Usually it's referred to as just a "button."</i> For example, when you press the "Search" button on the ShareLane home page, you send whatever was typed into the text box on the left of the Search button. A common problem with the Submit button is entering the wrong value of the attribute <code>value</code>.</p>
Reset button	 <pre><input type='reset' value='Reset'></pre>
	<p>The Reset button returns the values inside the Web form back to their defaults.</p>
Example 1:	
<p>Here is a UI once a particular Web page is opened:</p> 	
<p>As you can see, the option "Play with kids" is checked by default. If we select the button "Go fishing" and then press the "Reset" button, the selection will automatically shift back to the "Play with kids" option.</p>	
Example 2:	
<p>Here is another Web page:</p>	

Go fishing
 Play with kids
 Repair dishwasher
Reset

As you see, the default setting here is that none of the radio buttons are selected. If we select "Play with kids" and press the Reset button, then all the radio buttons will automatically become unselected (another term is *cleared*); i.e., they'll return to their default state.

I hope that this little intro to Web elements was useful! Now, let's get back to the BTS attributes.

ATTACHMENT

In some cases, it may take a long time to write a good textual Description of the bug (especially when the tester has to give details about what happens on the UI). It all depends on concrete situation, but as a rule, one of the best solutions is to **illustrate** the bug with an attachment.

Do this exercise (instructions are for Windows OS):

1. Go to Test Portal>More Stuff>Basic Web page elements.
2. Press the keyboard button *PrtScrn* (or whatever the abbreviation is on your keyboard for "Print Screen"); this button is located on the upper right side of a standard keyboard.
3. Now, go to Start/Programs/Accessories/Paint.
4. Do **Ctrl+v** on your keyboard; i.e., press the "v" key while holding down **Ctrl**.
5. Save the file with the extension **.jpg (.jpeg)**.

This is how you create an image that can become an attachment to the bugs in the BTS.

BTW

After you copied the clipboard content into the Paint program, you can

- Select->cut->save in the file precisely the area where bug is and/or
- Put an arrow to (or a circle around) a certain area to attract attention.

SL

See Attachment for Bug #1 in the Bug Vault.

BTW

If the attachment **completely** explains the bug, then it's okay if your Description consists of only one phrase: "See attachment."

I'd also recommend **ALWAYS** putting the phrase "see attachment" into the Description if there is an attachment. Why? Just to attract attention, because whoever views the BTS page with the bug Description can simply miss the fact that an attachment is available.

Please note that attachments are not restricted to images. You can also attach other types of files (e.g., text files with logs, etc.). The idea is to provide the best possible illustration of the bug.

SUBMITTED BY

The BTS automatically fills in this attribute with an alias of the person who filed the bug. The bug can be filed by any person who has an account in the BTS *and* the bug filing privilege – i.e., the tester, developer, PM, etc. This attribute's value is not editable.

DATE

The BTS automatically fills in the value of this attribute with the precise time the bug was filed. This attribute's value is not editable.

ASSIGNED TO

Here we come to the very important concept of the bug owner. The **bug owner is the concrete person responsible for the next step towards bug closure.**

The “Assigned to” drop-down menu is needed to specify an alias of the bug owner. When filing a bug, the tester should always select from *Assigned to*:

- A. The alias of the appropriate developer (e.g., "Billy")

or

- B. The tester's own alias

A. How do you know who the "appropriate developer" is? In the majority of cases, it's the developer who wrote the buggy code. But what if he or she has left the company (or the world), or moved to another team inside Development, so he or she is not responsible anymore for fixing that particular bug? In that case, we can "take our business" to

- Either the person currently responsible for that functional area (e.g., "Search")

- Or to the person inside Development who would **reassign** the bug to an appropriate developer.

That person is usually the dev manager.

The logical question is this: How do we get information about current responsibilities? You can ask questions and get answers, but it's not the best solution here. The best solution is to create and **maintain** two very important documents: **Who Does What** and **Who Owns What**. *Needless to say, these documents should be posted on the Wiki.*

Who Does What relates to **current projects** (a separate Who Does What can be created for each major release.)

Who Owns What relates to **areas of expertise**.

Example: Who Does What

Spec ID	Spec name	Product Manager	Developer	QA
1298	"Security Improvements to sign-up flow"	Lisa Wu	Roger Staunton	James King

Example: Who Owns What

Area	Product Manager	Developer	QA
Sign-up	Lisa Wu	Roger Staunton Victor Brown	Alex Young James King

You can get templates from Downloads on QATutor.com.

It's critical to keep these two documents up-to-date and fill them up with the correct info.

If a company consists of 3 persons, it's more or less clear who does what and who owns what, but once a start-up begins to grow, there are more people, more projects, more communication (and more miscommunication), more things to do, more information to digest, etc. It becomes very important to have **centralized sources of information** about current projects and areas of expertise. It's not only about the *Assigned to* attribute. It's about keeping track of things and knowing where to go for help. I strongly recommend both of those documents.

B. In some cases (especially in a start-up mess), the tester has no idea who will be fixing the bug. But that "no idea" should not prevent the tester from filing the bug report! The tester just needs to select his or her name from the *Assigned to* list, file the bug and then find out who is "Da Man" or "Da Woman" responsible for fixing the bug. I know of a situation where on a daily basis the dev manager would simply go through the list of freshly filed bugs assigned to testers and change the *Assigned to* values to the names of the appropriate developers.

Below are several BTWs related to *Assigned to*.

BTW

In many cases, programmers file bugs against themselves. This is not masochism. Programmers do this because:

- The BTS is a convenient way to keep track of bugs
- They need to justify time spent for bug fixes with an argument like: "I spent all Wednesday fixing bugs #232, #234, and #237!"

BTW

In some cases, programmers play ping-pong with bugs by reassigning bugs to each other with comments like, "This is not my bug" -> "Not, it's your bug"->"No, it's your bug"->"No, you should fix it", etc. The result of this game is always a delay in bug fixes.

BTW

People work for companies, and they leave companies. It's life. However, after they leave, their BTS accounts should be disabled but NEVER deleted. Deleting an account from the BTS can screw up the data integrity and create a real mess.

ASSIGNED BY

The value of Assigned by is the alias of the person who assigned the bug to its current owner. As is the case with Submitted by, this is an automatically populated attribute. This attribute's value is not editable.

VERIFIER

The Verifier is a drop-down menu with the same list of aliases as *Assigned to*. The Verifier is the person who must regress the bug once it is fixed. By default, the value of the Verifier is the alias of the person who filed the bug (Submitted by).

Remember, a bug can be filed by anyone who has an account in the BTS and a bug filing privilege; e.g., by the PM. Of course, it's not the responsibility of the PM to regress the bug. So, if the PM (or another non-tester) files a bug, then that person simply selects the tester's alias from the Verifier menu and lives happily ever after without thinking about bug fix verification. BTW, that "happily ever after" *can* be disturbed, because sometimes a bug is returned to the bug filer – e.g., if the assigned developer cannot reproduce the bug. Read on!

BTW

Every account in a professional BTS usually belongs to a concrete group. As a rule, there are at least 3 groups:

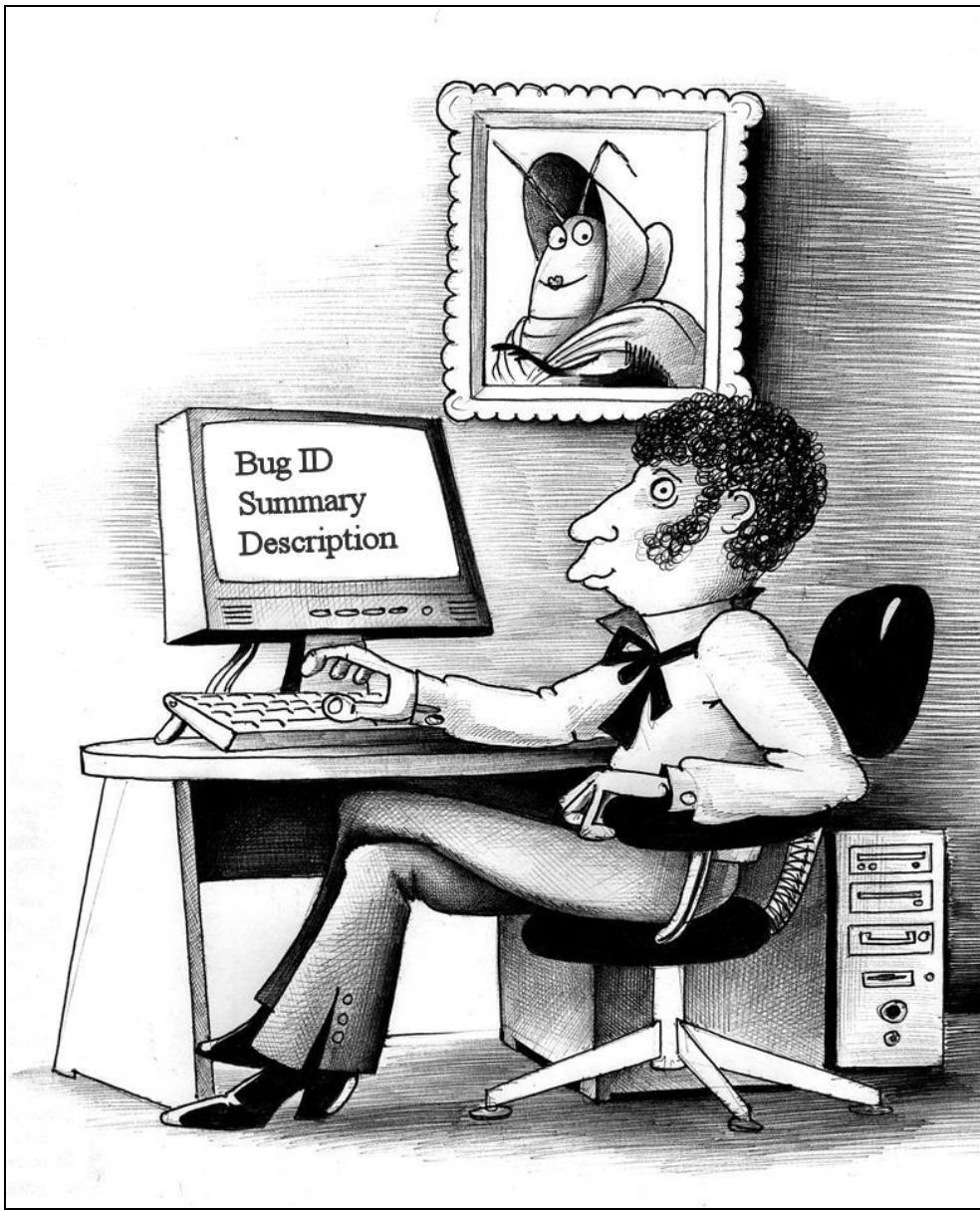
Testers
Programmers
Others

Each group is associated with a set of privileges. For example, as a rule, only the person who belongs to the Testers group can close bugs.

BTW

Professional BTSs allow for neat things like making a bug filer select a tester's name from the Verifier menu if the bug filer doesn't belong to the Testers group (otherwise a bug cannot be submitted).

There are many nuances about BTS privileges that we will not cover here. We have much more important things to learn and to do.



COMPONENT

As a rule, this is a drop-down menu with a list of the functional areas of the Web site. For example, at ShareLane we have this list of components:

- Registration
- Search
- Shopping Cart
- Checkout
- Other

The benefit of having this attribute is that we can sort bugs by their Component values.

FOUND ON

This is drop-down menu with a list of the environments where the bug was found. At ShareLane, we also added Spec to the list for spec bugs:

Production
main.sharelane.com
old.sharelane.com
Other environment
Spec

Of course, the worst thing is when a bug is found in production.

Here are the benefits of having this attribute:

- It's clear where to reproduce the bug.
- Bugs can be sorted by Found on values.

VERSION

This is the release version of the environment where the bug was found; e.g., if the application version is 1.0-23/34, we put 1.0.

BUILD

This refers to the build number of the environment where the bug was found; e.g., if the application version is 1.0-23/34, we put 23.

DB

The DB schema version of the environment where the bug was found: e.g., if the application version is 1.0-23/34, we put 34.

COMMENTS

This is multiuse text area. This attribute has 2 main usages:

- **Comments about the bug itself;** e.g., give more info about the steps to reproduce.
- **Comments about bug-related actions;** e.g., when a developer reassigns a bug to another developer.

As a rule, Comments is an optional field, but in some cases it makes sense to make it required – e.g., when the developer cannot reproduce the bug and reassigns it back to whoever submitted the bug.

SEVERITY

This is a drop-down menu with values S1-S4 inclusively. **Severity is the magnitude of the bug's impact on the software.** Severity is a **technical characteristic**. In order to determine Severity, we must find a match between

- The Severity category
and
- The type of bug's impact

The document we use for this is **Bug Severity Definitions** (see QATutor.com Downloads for the template).

Here is an example of the document (explanations will follow):

Severity	Type of impact
S1	<ul style="list-style-type: none"> - crash - data loss - security issue
S2	<ul style="list-style-type: none"> - site hangs - blocking issue
S3	<ul style="list-style-type: none"> - functional issue
S4	<ul style="list-style-type: none"> - other

Details:

Severity	Details and examples
S1	<p>CRASH</p> <p>This is situation when a Web site is completely nonfunctional because of a software problem.</p> <p>DATA LOSS usually happens in the following cases:</p> <ol style="list-style-type: none"> 1. Data is lost halfway to the DB 2. DB is populated with partial data 3. Data is deleted from the DB <p>Example 1: A user submits his or her data via the registration form <code>register.py</code>, but the module <code>db_lib.py</code> has a bug in the function <code>update()</code> and the data is not inserted into DB.</p> <p>Example 2: A good example is the earlier situation when the <code>first_name</code> column in the DB table <code>users</code> couldn't hold 51st+ character while the first name text field allowed the user to type in 51+characters. Of course, this was an extreme example; not many people have a first name with 51+ characters. In reality, data loss of this type usually happens when a programmer allows an unreasonably small capacity for the DB column; e.g., <code>Varchar(5)</code> – only 5 characters for the <code>first_name</code>.</p>

	<p>Example 3: This may happen when the application core file mistakenly deletes/updates data. For example, in 2.0 we introduced the “Update user info” functionality. Willy wrote buggy code that would update all the records in the DB table <i>users</i> instead of updating only one record: precisely, the record of the user who is changing his or her info. So a bug resulted in this situation: if a user changes his last name to "Lee", the last name of every ShareLane user becomes "Lee".</p> <p><i>BTW, bugs of this kind are another reason to set up daily DB backups in both the testing and production environments!</i></p> <p>Please note that data loss might not be related only to the DB; e.g., because of the bug the content in the log file can be <i>overwritten</i> with new lines instead of being <i>appended</i> with new lines.</p> <h3>SECURITY ISSUE</h3> <p>Example: Because the bug site that accepts credit cards stopped using a secure connection between the Web server and the Web browser, unencrypted credit card data can be discovered by special software and stolen.</p>
S2	<h3>SITE HANGS</h3> <p>This is usually a performance issue; e.g., it takes a long time to load the Web page.</p> <h3>BLOCKING ISSUE</h3> <p>The term "blocking issue" has a wide meaning. It refers to a bug that blocks:</p> <ol style="list-style-type: none">1. Development2. Testing3. Site usage <p>Example 1: Willy is supposed to write functionality A. Functionality A should use the Python classes from Billy's code. But as it appears, those classes have a number of serious design bugs. Therefore, it just doesn't make sense to start working on functionality A until Billy has fixed those bugs. So Willy is blocked from coding.</p> <p>Example 2: Willy wrote functionality A. Billy wrote functionality B. B depends on A, but A doesn't work. Thus, tester Rajeev cannot test functionality B. In other words, Rajeev's testing is blocked.</p>

	<p>Example 3: Users are blocked from buying books on ShareLane because Checkout doesn't work.</p> <p>A very important term that is associated with a blocking issue situation is workaround. For example, if the Account Creator (Test Portal>Helpers>Account Creator) used by testers for automated account generation doesn't work, it's not an issue that blocks testing because new user accounts can be created manually. So, in that situation, manual account creation is a workaround. If there is a reasonable workaround, then an issue cannot be considered as blocking.</p>
S3	<p>FUNCTIONAL ISSUE</p> <p>This category is for all functional bugs that don't fall under S1 or S2. It's no wonder that majority of bugs will be S3. That's why S3 is, as a rule, the default Severity when testers file a bug.</p>
S4	<p>OTHER</p> <p>This includes all bugs that don't fall under S1, S2, or S3. As a rule, here you'll find UI bugs (e.g., layout problems, spelling, etc.).</p>

PRIORITY

This is a drop-down menu with values P1-P4 inclusively. **Priority is the magnitude of a bug's impact on the company's business**. There is often confusion between Priority and Severity, so let's point out the differences:

Severity refers to the **technical** aspect of a bug.

Priority refers to the **business** aspect of the bug.

In other words, **Severity uses technical criteria to grade the bug, while Priority uses business criteria**. That's why it's almost always clear which severity to assign to the bug, while the Priority of a concrete bug is often the subject of arguments and political games.

Example

If we have ANY kind of data loss, the severity must ALWAYS be S1, because we **firmly associate S1** and data loss. Let's recall the bug we discussed when a global update happened to DB table users during every change in the user profile (our recent example when the last name of every user became "Lee"). The Severity of the bug is S1 – no argument about that. Let's find out about Priority.

Tester Anitha assigned a P1 (major issue) to this bug – we are talking about a HUGE data loss. But once the bug was filed, Willy spent about 10 seconds fixing it, and then he set its Priority as P4. When Anitha asked Willy what was going on, Willy explained that the bug existed only under 1 condition: when a certain constant in the DB was set to a certain weird value – exactly the value that was erroneously set by a tester, George, when he hacked into the DB of main.sharelane.com while performing component testing on the discount functionality for big spenders. Production would *never* have that problem.

Does Anitha's logic about P1 make sense? Yes, it's no joke when THAT kind of thing happens, even in the test environment...even under some stupid, unreal condition.

Does Willy's logic about P4 make sense? Yes, because production will never have this issue.

The moral of the story:

- There is usually a unanimous consensus about Severity.
- There can be **opposite, yet legitimate**, arguments about Priority.

Do you remember the quote from the great physicist Niels Bohr (during our lecture about test cases)?

"The opposite of a correct statement is a false statement. The opposite of a profound truth may well be another profound truth." This quote is a key to understanding why bug Priority is often a subject of controversy.

The best tool that helps to reduce any controversy associated with assigning Priority is called the **Bug Priority Definitions**. Please look at the example of Bug Priority Definitions below (you can get the template from Downloads on QATutor.com).

Priority	Types of issues	Examples
P1 (Critical)	1. Critical reliability/performance issue* 2. Critical legal issue* 3. Critical business issue* 4. Critical security issue* 5. Core functionality** is not working; no workaround	For 1: Site is dead For 2: Legal suit is filed (or about to be filed) against the company Court order For 3: Integration with partner's site is broken For 4: Intruder's backdoor is discovered For 5: 500 - Internal Server Error when user clicks Shopping Cart link
P2 (Major)	1. Major reliability/performance problem* 2. Major legal issue* 3. Major business issue* 4. Major security issue* 5. Core functionality is not working; workaround exists	For 1: Unreasonably long time to load a Web page, e.g., 10 seconds For 2: Age verification during user registration was not implemented For 3:

		Inaccuracies in banner rotation For 4: Under certain conditions, name and address of another user is displayed on Checkout page For 5: Cannot complete purchase if card expiration date is this year (workaround is to use another card).
P3 (moderate)	1. Unpleasant but bearable reliability/performance issue* 2. Legal issue of moderate importance* 3. Business issue of moderate importance* 4. Security issue of moderate importance* 5. Functionality doesn't work (cases that don't fall into P1, P2, P4 categories)	For 1: Intermittent delays loading a Web page For 2: Copyright issue with one of the books offered for sale at ShareLane For 3: Show Book page is missing info about the publisher For 4: Log out button doesn't work in some browsers For 5: Cannot search for keywords that consist of 3 characters or less
P4 (Low)	Cosmetic issue* of low importance that doesn't affect reliability/performance, or legal, business, security sides of company operation Low importance/probability functional problems	Typos Problems with layout

* Issue can be present or potential
** Core functionality is the functionality that is essential for the book sale, e.g., Shopping Cart or Checkout.

There are 2 more very important documents that are closely associated with Bug Priority Definitions.

1. The **Bug Resolution Times** document defines concrete deadlines for the programmer to fix the bug. Deadlines depend on Priority.

Example

Bug Priority	Production	Test Environment
P1	ASAP. Drop whatever you're doing.	24 hours
P2	24 hours	48 hours
P3	n/a*	72 hours
P4	n/a**	Fix if possible

* P3s are to be fixed in the next minor or major release

** P4s are fixed if possible and thus are not suitable for patch releases.

2. The **Go/No-Go Criteria** document is a set of conditions under which the release goes to production. As a rule, P1s and P2s must be fixed before releasing code to production. So, in some cases P2 bugs are changed to P3s to meet the criteria (talking about political games!).

You can find both Bug Resolution Times and Go/No Go criteria under Downloads on QATutor.com

BTW

Before the Go/No-Go meeting, the tester must print out several copies of the BTS report with any open bugs (for the release that is going to be discussed at the meeting) and refresh his or her memory about the situation surrounding each of those bugs.

At beginning of the meeting, the tester distributes these reports among the participants.

During the meeting, the tester must be able to communicate intelligently about each of the bugs. So come prepared!

BTW

If you, as a tester, have a firm position regarding one Priority and a programmer has a firm position regarding another Priority for the same bug, take the argument to your manager. This will be more productive than changing the Priority back and forth.

Always remember that **Priority is a powerful tool that affects the developer's schedule**, so do your best to assign a Priority as close to the Bug Priority Definitions as possible.

To conclude our talk about Priority let me give you an important BTW.

BTW

In one of his great presentations, QA patriarch Cem Kaner describes how testers can “sell” bugs to developers to make them fix those bugs (you can find this presentation on www.kaner.com).

Regretfully, this is often a reality in a tester’s work, but this is only a reality IF appropriate processes are not in place, or IF appropriate processes are introduced but not enforced.

Example

If, according to the **Bug Resolution Times** document, a P1 bug must be fixed within 24 hours, it MUST be fixed within 24 hours. If it's not fixed within 24 hours, the dev manager MUST have a one-on-one meeting with the responsible developer. If that developer doesn't care that a P1 bug blocks testers, his or her ass should be fired. Of course, life is life and sometimes it's really hard to follow the process, but "sometimes" happens only occasionally, and in the majority of cases, it's the direct fault of the developer if a bug is not fixed in time. If the developer doesn't have enough time to fix the bug (for whatever reason, like too much other coding to do), he or she **must raise a red flag** telling his or her manager, "No can do," (rather than hope that no one notices that a bug is not fixed.)

Nobody needs bureaucracy: it's harmful for the team morale and for the business, and a company must be really careful which processes and standards to adopt. BUT once they are adopted, there shouldn't be any "nobody has noticed it" situations; i.e., situations where the person who screwed up isn't asked the simple question, "Why didn't you follow the rules?"

Apart from processes being introduced and enforced, there is the great concept of VALUE that test engineers bring to developers (and not only to developers!). The idea is simple: **If you are valuable to developers, they respect you and try to help you.**

What do YOU value in your colleagues? Honesty, a positive attitude, a desire to help, politeness, hard work, care for the company, and experience...surely, others will appreciate those things too if they see them in you.

The things that I just listed do include experience, but **experience** (although it's a really important part) **is in reality just a fraction of what it takes to be a real professional**. Even if you are a beginner tester, you can create value for yourself by showing quality aspects of your personality, and in place of experience you will exhibit a "passion to gain experience." **In the majority of cases, your personality means more than your experience.**

During an interview, you might be asked the question: "A certain programmer didn't fix his/her bug. What would you do?" Now you know the answer. You should say: "First, we should introduce the proper procedures to prevent that situation from happening, because then the developer would know the exact deadline for the bug fix. Second, I'll do my best to create value for myself, and the developers will want to assist me."

ALSO NOTIFY

This is usually a text box where you can put the aliases of people who, in your opinion, would be interested to get emails containing:

- Notification that the bug was filed
- Updates about changes that happened to the bug

BTW

As a rule, by default, **when a bug is filed**, an email is sent to the bug filer (Submitted by) and to the person who will fix the bug (*Assigned to*).

While testing code that has been written according to a concrete spec, I usually put the PM's alias into the Also Notify list. PMs appreciate this gesture.

CHANGE HISTORY

This is an automatically populated field in case any events happen to the bug:

- The fact of bug filing
- Any change to the bug

The Change History usually includes:

- Date and exact time (to the second) of the bug filing/change
- Alias of the person who filed a bug/made a change
- Fact that the bug was filed, or what was changed and how it was changed

Therefore, whatever someone changes about the bug is automatically logged, and that log (called Change History) is available to everyone who has a privilege to view bugs in BTS.

TYPE

This is drop-down menu with two values:

- Bug
- Feature Request

We select "Bug" if it's about a bug. We select "Feature Request" if we want to suggest and/or keep track of some kind of non-bug-related improvement. Please, note that EFR (Emergency Feature Request) is usually filed with type "Bug".

STATUS

This is drop-down menu with 3 values:

- Open
 - Closed
 - Re-open
-
- Open is automatically assigned after a bug is filed.
 - Closed must be selected by the tester when he or she closes the bug.
 - Re-open must be selected once the bug is back. (Read on...)

When do we have a situation where we need to re-open a closed bug?

Example

Situation 1: A programmer changed one piece of code and broke another piece of code that was fixed earlier, so the bug was **reintroduced**.

Situation 2: A programmer changed the code in the production branch but forgot to commit the changed code into the trunk, so after the new branch was cut off from the trunk, the bug showed up in the new branch.

BTW

ALL found bugs must be filed into the BTS – no excuses and no exclusions. It is your job to file bugs. Yes, bugs can be communicated verbally and/or via email, IM, or whatever – communication is a great thing, and you have to do your best to assist a programmer to understand the bug. But the main thing is that YOU MUST ALWAYS FILE A BUG. You can file the bug right after you find it or you can go to the developer first, but the rule remains: ALL FOUND BUGS MUST BE FILED INTO THE BTS.

If the programmer or PM doesn't understand that bug filing is a part of your profession, try to explain it to them, or send them to your manager. No sane programmer or PM would ever take bug filing personally. They might make reasonable claims that the bug Description sucks or that the Priority is wrong, but I've never seen a real professional who would get offended by the FACT that a bug was filed.

BTW

Bugs should never be removed from the BTS. Period.

RESOLUTION

This is a drop-down menu with the following list:

*Reported
Assigned
Fix in progress
Fixed
Fix is verified
Verification failed
Cannot reproduce
Duplicate
Not a bug
3rd party bug
No longer applicable*

Resolution is one of the most important attributes. If Status is about global things like "was born," "died," and "got reincarnated," Resolution provides details like "graduated from college", "got married", "bought a condo" etc. Resolution details the stages of a bug's life.

Question: What drives a bug from one life stage to another?

Answer: Certain activities geared towards bug resolving.

Question: Where can we find information about those "certain activities" and the associations between them and the concrete values for resolution?

Answer: In the BTP.

Let's learn about each Resolution. BTW, because the majority of bugs are found in the software (not in the specs or other documentation), we'll be talking about situations where the person responsible for the bug fix is the programmer.

Reported

This Resolution must be chosen when the person who files a bug doesn't know who will be fixing the bug. This is the situation when a tester assigns a bug to himself or herself.

BTW

In case of sophisticated BTS software, we can program our BTP into the BTS; e.g., depending on the situation, the required Resolution value would automatically be selected. For this course, I assume that the people who use the BTS voluntarily follow the BTP.

SL *Bug Vault>Bug #2*

Assigned

This Resolution means that the programmer in *Assigned to* should investigate the bug.

Example 1: The person who files a bug knows who is going to fix the bug, so that person selects name of the programmer from *Assigned to* and selects the Assigned for Resolution.

SL *Bug Vault>Bug #3 – please note that this bug was invented to illustrate the point. ShareLane app. doesn't really have a bug like this.*

Example 2: The person filed a bug with Reported status, but changed Reported to Assigned (and put programmer's alias into *Assigned to*) once he or she found out who was going to fix the bug.

SL *Bug Vault>Bug #4 (see Change History)*

Fix in progress

The programmer selects this Resolution as soon as he or she starts working on the bug fix.

SL *Bug Vault>Bug #5 (see Change History)*

Fixed

The programmer selects this resolution as soon as he or she has checked the bug fix into the CVS. Along with that, the programmer must select the same alias as in Verifier from the drop-down menu *Assigned to*.

SL *Bug Vault>Bug #6 (see Change History)*

BTW

Testers should remember that it takes time before the build script picks up the CVS content and populates it onto a particular environment. So, if a programmer changed the Resolution to Fixed at 8:00 am, and the next build starts at 9:00 am and takes 15 minutes to complete, then the bug fix will be available on the target environment no earlier than 9:15 am.

The Moral of the Story: Before you start regressing the bug, make sure that the build with the bug fix has already been pushed to the target environment and that the build was successful.

SL In other words, check out 2 things before you begin bug fix verification:

- **Build status** (see example at Test Portal>Release Engineering>Build Status)
- **Application version on the target environment** (you can see it if you view the HTML source of any Web page on ShareLane.com: <!-- application version 1.0-23/34 -->)

Fix is verified

Remember that bug regression consists of two parts:

1. Verification that the bug was really fixed
2. Checking if the bug fix hasn't broken some other part of the software.

First of all, we just try to reproduce the bug following the instructions in the Description:

- If the bug is NOT reproducible, then it was really fixed.
- If bug IS reproducible, then it wasn't fixed, so we send it back to the developer with the Resolution "Verification failed" – and we DO NOT execute Part 2 of the bug fix verification.

If the bug is NOT reproducible, we move to Part 2. This is where it gets interesting. In the case of more or less complex software, it's sometimes extremely hard (even for a programmer) to confidently predict how a certain change to a certain piece of code will affect other parts of the software. So, the only way to do comprehensive regression testing is to perform 100% coverage of all possible scenarios (what's usually impossible!).

The standard way to execute Part 2 is to perform a basic end-to-end test of the feature that contained the bug. In special cases (e.g., in case of EBF) you can:

- a. Ask the developer who fixed the bug his or her opinion of what could have gotten messed up as a result of the bug fix and what he recommends that you check out in particular.
- b. Next follow his instructions.
- c. Then perform a basic end-to-end test of the feature that contained the bug.

Please note that you can:

- Change the Resolution from "Fixed" to "Fix is verified" AND
- Close the bug (change its status to Closed) as soon as Part 1 is finished and you have verified that the bug was really fixed.

Why bother about Part 2? We need Part 2 simply as a safety measure. Of course, in the case of Part 2 we're not talking about serious testing; we just do a quick check that usually takes several minutes. What happens if we find a bug? Simple – we file new bug (or re-open closed bug).

SL *Bug Vault>Bug #7 (see Change History)*

Verification failed

The verifier changes the Resolution to "Verification failed" and "Assigned to" to the alias of the responsible developer if the bug is reproducible; i.e., if Part 1 of the bug fix verification failed.

SL *Bug Vault>Bug #8 (see Change History)*

Cannot Reproduce

This unpleasant situation occurs if the developer tries to reproduce the bug assigned to him or her, and cannot do it. The bug is usually not reproducible for 1 of 3 reasons:

1. The tester didn't provide a comprehensive Description.
2. The tester's environment and the developer's environment are different.
3. There is no bug.

Let's look at these reasons one by one.

1. The tester didn't provide a comprehensive Description.

One of the main concepts about bugs is the idea that **if a bug exists, then it is reproducible**. NEVER file a bug until you reproduce it at least once after you discovered it. Read the following *Brain Positioning*, and remember it for the rest of your testing career!

Brain Positioning

Sometimes you can get really excited once you find a bug, especially a fat P1, and you can be tempted to run to the developer RIGHT AWAY to share your finding. Don't do this. Whatever the problem is...

1. Try to reproduce it.
2. Try to narrow it down; i.e., isolate the problem.

For example: Let's assume that after you clicked the "Make Payment" button using Internet Explorer 6 under Windows XP, you got a "500 – Internal Server Error". What you should do next is this:

1. Try to reproduce it again, and write down the exact parameters (book title/price/quantity/credit card info: type: number, cvv2, expiration date).
2. "Play" with those parameters; e.g.; try to reproduce the bug using a different credit card.
3. Try to use a different browser/OS combo; e.g. try Firefox under Windows XP or Safari under Mac OS.
4. If a feature is on production and you found a bug in the test environment, try to reproduce the bug on production.

After your 4-step investigation is done, file a bug report right away, or go talk to developer (and file the bug later).

Again, if a bug exists, then it's reproducible. Each concrete bug is a resulting combination of a certain scenario, i.e.,

1. Actions
2. Data
3. Conditions.

Usually people are more focused on providing info about actions and data rather than about conditions. Try to avoid that pitfall! In many cases, it's the condition that makes all the difference. In some cases, a condition can be very unusual or hard to grasp, but if you pay attention you'll find that condition. Please read the story below to better understand my point.

4 Scientists and 1 Flask

Once upon a time, there was a pharmaceutical laboratory where 4 scientists worked to find cures for illnesses. Scientist Leo N. invented a unique chemical substance that could serve as the basis of a new, powerful medicine. The problem was that the 3 other guys could not produce the substance, even if they methodically followed Leo's every step. Leo was happy to share all the information he possessed about the process, but the others had no luck - it seemed like Leo had some unexplainable ability. One evening, those 4 scientists with their PhDs in chemistry got together and decided that they were going to believe in miracles, but only after one last thing: during the preparation of the substance, Leo's EACH AND EVERY action must be captured on video and analyzed afterwards.

Following the plan, after the video was ready, the 4 colleagues got together and made a thorough analysis of EACH AND EVERY one of Leo's actions. After several hours of analysis and conducting tests they found out what was really happening: in the middle of the preparations, whoever was preparing the substance had to walk for 1 minute between the two laboratories, which were situated in different buildings. It was wintertime, BTW. Leo was a smoker, so before going outside **he would put the flask under his coat to free his hands for a cigarette and matches**. Therefore, the substance in the flask wasn't exposed to the cold like it was with the 3 other scientists who didn't smoke and who simply ran between the buildings with the flask in their hands! So the magical condition that made all the difference was: **Don't expose the flask to the cold!**

The moral of the story here is that, in some cases, even a little, hard-to-notice nuance makes all the difference.

Let's get back to our testing. In many cases, a bug is not reproducible not because the tester didn't know the scenarios/conditions, but because he or she just didn't write a comprehensive Description.

Brain Positioning

Please remember that even if you get excited about finding some buggy area and you want to continue hunting for bugs instead of filing bugs, it's your direct responsibility to:

- File bugs
- Make sure that others can understand your bugs and be able to reproduce them

It really sucks when a developer must put off his or her coding, spend time trying to reproduce a bug, and is not able to do it because the tester forgot to include a little detail right in the middle of "Steps to reproduce". You must avoid situations like that! **Do your best to avoid "merchandise returns" that come with the Resolution "Cannot reproduce"!**

2. The tester's environment and the developer's environments are different.

Again, it's all about conditions. There are situations where some kind of underlying item is different in the dev environment (e.g., billy.sharelane.com) than in the test environment (e.g., main.sharelane.com). For example, the dev environment could have a more recent version of the Python interpreter, so Billy could use the Python libraries that exist in his environment but are missing on main.sharelane.com. The tester sees a bug and files it, only to get a "Cannot reproduce" Resolution, which leads to a waste of time going back and forth with Billy trying to understand what's really going on!

3. There is no bug.

Here's a potential situation: The DB on main.sharelane.com is down for maintenance, and the tester doesn't know about it. He or she sees the bug, tries to reproduce it (with success), and then files it. The

programmer tries to reproduce the bug **when the DB outage is over**, but the bug doesn't exist anymore. So the programmer sends the bug back with the "Cannot reproduce" Resolution. *BTW, a similar unpleasant situation might happen when the testing is done while a push of a new build is under way. ALWAYS check the build status page before starting testing!*

In case of a "Cannot reproduce" Resolution, whoever selects that Resolution should also assign the bug back to the person who filed it; i.e., the bug owner must be a person from the "Submitted by" field.

What happens if a tester receives a bug back? He or she should either close it OR provide more details and assign it back to the developer with an "Assigned" Resolution.

SL *Bug Vault>Bug #9 (see Change History)*

Duplicate

This Resolution is selected if another bug was already filed for the same issue. In software companies, the BTS may contain thousands of open and closed bugs, and sometimes it's not physically possible to review each of them to avoid filing a duplicate. The best way to avoid duplicates is to keep track of:

- All **new** bugs OR
- New bugs that meet certain criteria. For example, bugs found in particular area, e.g., "Checkout"

A professional BTS usually allows you to modify the settings for your BTS account and sends you an email if any new bug is filed or a filed bug meets certain criteria. That way you'll know what was filed and what wasn't.

On the other hand, there can be not-so-obvious situations. For example, **two or more filed bugs can be the result of the same root cause**.

Example

SL Look at Bug #4 in the Bug Vault: "*shopping_cart.py: 2% discount if user buys 12-19 books inclusively*".

The WRONG way to do it is to file 8 bugs:

1. "*shopping_cart.py: 2% discount if user buys 12 books*".
- ...
8. "*shopping_cart.py: 2% discount if user buys 19 books*".

The root cause is the following statement in shopping_cart.py:

```
if q >= 12 and q <= 49:  
    discount = 2
```

So, one bug is enough to cover all 8 cases.

Black box testers don't look into the code, and situations like this do take place. So when the developer returns your bug with a "Duplicate" Resolution, explaining (in Comments) that your bug was caused by the same code as another bug, don't take it personally - that happens, and it's okay.

Once a bug is marked as Duplicate, the person making that change must also put the original bug ID in Comments and assign the bug back to the person who submitted it.

SL *Bug Vault>Bug #10 (see Change History)*

Not a bug

Start-ups, especially in the early stages, are usually a mess.

In some cases, it's really hard to find out the **correct** expected result, and our old friends: common sense and life experience might be different from what the PM tried to communicate during his or her talk with the developer some time ago.

In some cases the PM communicates a product change to the developer and forgets to:

- update the spec and
- mention that product change to the tester.

In some cases, the tester is correct that it's a bug, but the developer thinks that it's a "Feature" and sends the bug back to the tester.

So be prepared to answer the question: "Why is that a bug?" If you were wrong about it, but you had reasons for believing that it WAS a bug, then there is no problem.

If you are not sure and it's a good time to ask ("Hey Linda, is this a bug or not?"), then ask. If nobody is available, then file a bug report. IMHO, it's better to file a bug (if you have reason to believe that it *is* a bug) and later find out that it's not a bug, than be scared of the "Not a bug" Resolution and ignore the real problem.

In any case, when you receive your bug back with the "Not a bug" Resolution, read the Comments and try to understand what the developer tried to communicate. If you still think you are right, go talk to the developer, and then:

- If you both agree that it is a bug, send it back to the developer (*Assigned to*: developer's alias; Resolution: "Assigned") with your explanations in "Comments".
- If you both agree that it's not a bug, just close it.

In some cases when a spec sucks, the tester and developer cannot come to a consensus about what the heck the PM meant. In that case, reassign the bug to the PM and let him or her decide whether it's a bug or not. In a situation like this, I'd also recommend filing a separate bug against the spec.

See an example of ping-pong with "Not a bug" here:

SL *Bug Vault>Bug #11 (see Change History)*

3rd party bug

The code in a software start-up cannot rely upon only in-house software, e.g., code written by the start-up's programmers. We use databases, compilers, interpreters, and Web servers that have been designed and developed by others. In many cases, we also integrate out software with the software of our partners, e.g., with credit card processors. So, here is the situation: you file a bug, the bug programmer returns this bug to you with a "3rd party bug" Resolution and the Comment that the bug is not in the software that he or she wrote, but instead in the software written by others. Here we might encounter two situations:

1. **We CANNOT influence "others" to fix their software** – e.g., if the bug is in the Python interpreter, we cannot call Python's father, Guido van Rossum and ask him to fix the bug before our release goes out. So, the only way to fix the bug is to find some programming workaround. So, if a programmer returns your bug with a "3rd party bug" Resolution, it doesn't solve the problem, because whether he or she likes it or not, help is not coming, and we have to find a solution ourselves. In a case like this, talk to the programmer and re-assign the bug back to him or her.

SL *Bug Vault>Bug #12 (see Change History) - please note that this bug was invented to illustrate the point. ShareLane app. doesn't really have this bug.*

2. **We CAN influence "others" to fix their software.**

SL For example, in checkout.py we have the function get_ccp_result() (see source code here: Test Portal>Application>Source code>checkout.py). This function connects to the payment processor to process credit card payments (*of course, in case of ShareLane there is no real processor - it's all training software*). That payment processor software belongs to another company, which is our service provider (also called "vendor"). If there is a bug in the vendor software, then we can call them and ask them to fix the problem ASAP. As a rule, the tester usually doesn't make this contact. As a rule, a contact like this is the responsibility of the project manager (PjM). *BTW, in start-ups PMs are often also PjMs.* So, if a developer returns a bug to you with a "3rd party bug" Resolution in this case, you should reassign the bug to the PjM and that person would be the bug owner who is responsible for resolving the situation.

SL *Bug Vault>Bug #13 (see Change History)*

No longer applicable

This is usually selected for bugs found in features that have been deprecated; i.e., features that are still available to users but no longer officially supported.

SL *Bug Vault>Bug #14 (see Change History)*

Bug Tracking Procedure

As you know:

- A bug can be filed by anyone who has a BTS account and bug filing privileges.
- At the time a bug is filed, the tester might not know who will fix the bug.
- Some feature requests can be filed as bugs with a "Feature request" type.
- The same bug can reemerge, so we can have a "Reopen" status.

However, for our educational purposes we'll consider the most typical BTP. Once you understand the typical model, you'll easily understand any variation of it. *BTW, a flowchart for a more complex BTP can be found under Downloads on QATutor.com, but please finish studying this lecture before looking at it.*

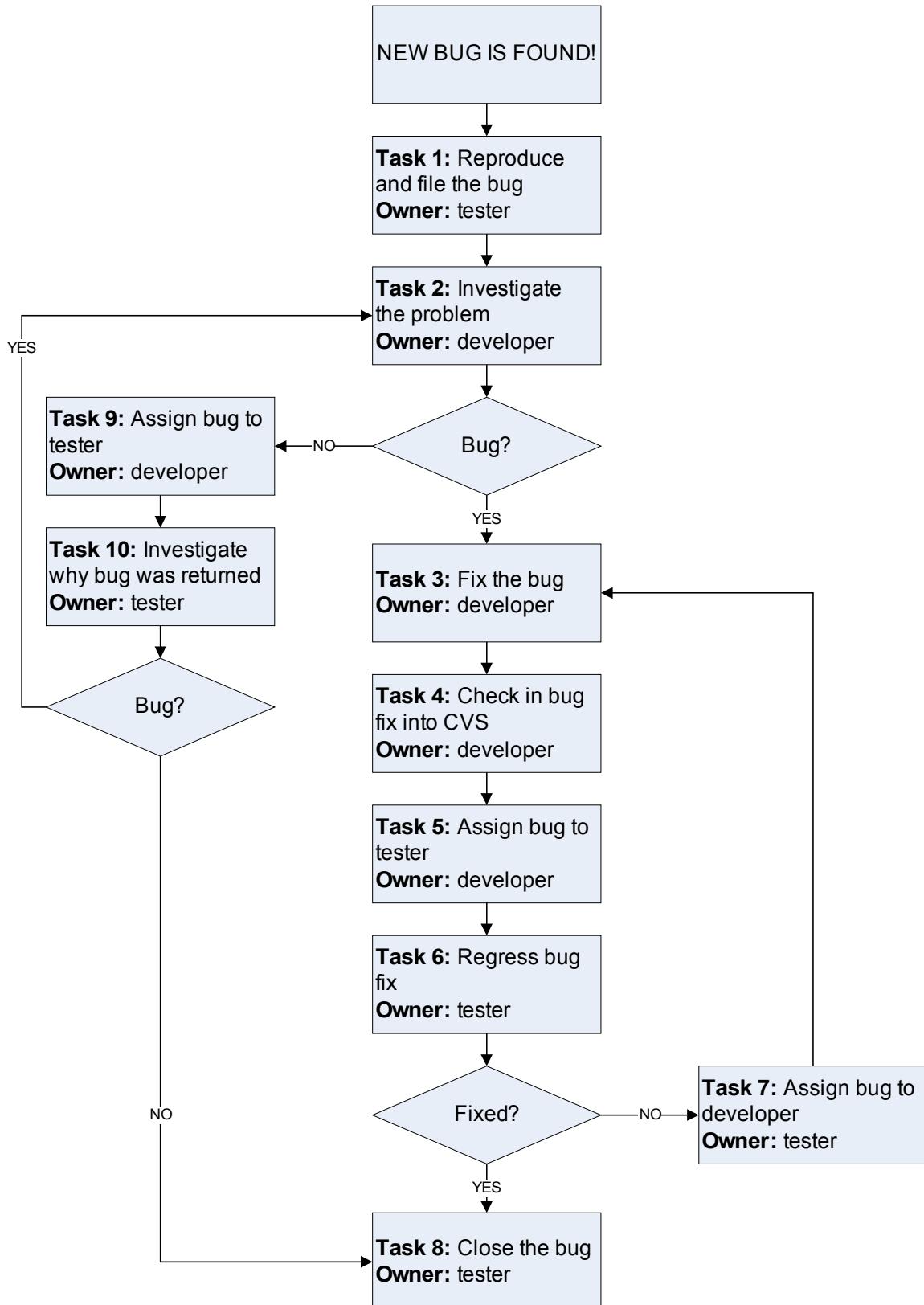
Let's make several assumptions:

- The bug is filed by a tester.
- The tester knows who will fix the bug.
- The bug is of the "Bug" type.
- The bug is new.

IMHO, the most comfortable way to describe the BTP (for educational purposes as well as in software companies) is to divide it into 2 parts:

1. BTP flowchart
2. Associations between the BTS Attributes and actions taken to resolve the bug

1. BTP flowchart (see next page)



2. Associations between the BTS Attributes and actions taken to resolve the bug

Task 1: After bug is found and the tester can reproduce it, the tester files it into the BTS.

BTS attribute	Value
Summary	Fill in with a brief summary of the problem
Description	Fill in with a detailed description of the problem
Attachment	If it makes sense, add attachment
Assigned to	Select alias of the developer who'll fix the bug
Component	Select name of the component where the bug was found
Found on	Select name of the environment where the bug was found
Version	Fill in with the software version, e.g., 1.0
Build	Fill in with the build number, e.g., 23
DB	Fill in with the DB schema version, e.g., 34
Severity	Select bug severity. Consult Bug Severity Definitions
Priority	Select bug priority. Consult Bug Priority Definitions
Notify	Fill in with the alias of the person to be notified about the filing and changes for this particular bug. Use comma delimited format to specify several aliases.
Type	"Bug"
Resolution	"Assigned"

Task 2: The developer tries to understand the problem.

IF IT'S A BUG (in the developer's opinion):

Task 3: The developer starts to fix the problem.

Resolution	"Fix in progress"
-------------------	-------------------

Task 4: The developer checks the bug fix into the CVS

Task 5: The developer assigns the bug back to the tester

Resolution	"Fixed"
Assigned to	Tester's alias

Task 6: The tester verifies the bug fix.

IF THE BUG FIX VERIFICATION FAILED:

Task 7: The tester returns the bug back to the developer.

Resolution	"Verification failed"
Assigned to	Developer's alias

ELSE:

Task 8: The tester closes the bug.

Resolution	"Fix is verified"
Status	"Closed"

ELSE:

Task 9: The developer returns the bug back to the tester.

Resolution	"Cannot Reproduce" "Duplicate" "Not a bug" "3rd party bug" "No longer applicable"
Assigned to	Tester's alias

Task 10: The tester tries to understand why the developer returned the bug.

IF IT'S A BUG (in the tester's opinion):

Task 2: The developer gets the bug back for further consideration.

Resolution	"Assigned"
Assigned to	Developer's alias

ELSE

Task 8: The tester closes the bug.

Status	"Closed"
--------	----------

Quick Closing Note About BTS And BTP

1. The actual steps to file a bug must be simple, easy to understand, well documented, and devoid of bureaucratic BS. What happens if it's hard to file a bug?

- Developers and testers will be reluctant to file bugs into the BTS and be inclined to communicate issues personally, via email, or IM, etc.

- Folks who are not developers or testers will not use the BTS at all. Instead they'll use email, MS Excel files, Post-it notes, etc.

2. At the beginning of the lecture I used a quote by my dear friend, computer scientist Daniel Kionka: "Sometimes you have to write down unwritten rules." The reason I did that is because often in start-ups many things are assumed, e.g., "P2 Priority can be attributed to this bug because, well, it *feels* like P2." The problem is this: **If we rely on individual assumptions for key things like the Bug Priority, we open the door to unproductive discussions and arguments.** In the long run, it's much better for everyone if we have clean, usable, documented standards. When we talk about bug tracking, the following 3 standards are a must:

- Bug Priority Definitions
- Bug Resolution Times
- Bug Tracking Procedure

For all 3 of the above standards the key concept is **effective simplicity**. It's not easy to develop solutions that are effective and simple at the same time, but once you develop them, your solutions (e.g., the BTS setup and BTP linked to the BTS) will work well for years.

Lecture Recap

1. The fact that a bug has been discovered has no practical value until we share information about that bug.
2. All found bugs must be filed into the BTS.
3. A bug can be filed by anyone who has a BTS account and bug filing privileges.
4. There are 2 stages of bug regression:
 - a. Verify that bug was really fixed (if bug was really fixed, you can close the bug right away).
 - b. Check to see if the process of fixing the bug has introduced new bugs.
5. On the one hand, the BTS is an **infrastructure** that enables us to

- create
- store
- view
- modify

information about bugs.

On the other hand, the BTS is a **communication medium** for SDLC participants.

6. The process that begins with filing a bug into the Bug Tracking System is called the Bug Tracking Process.

7. BTS Attributes:

- >**ID** is the unique identifier of each BTS bug record.
- >**Summary** is a brief synopsis of the problem.
- >**Description** is a detailed description of the problem.
- >**Attachment** is needed to illustrate the bug (e.g., with a screen shot)
- >**Submitted by** is the alias of person who has filed the bug.
- >**Date** is the date when the bug was filed.
- >**Assigned to** is required to specify the bug owner. Every open bug ALWAYS has a bug owner. The bug owner is the person responsible for next step in resolving the bug.
- >**Assigned by** is the alias of the last person who assigned the bug (i.e., whoever selected the new bug owner from the “Assigned to” drop-down menu).
- >**Verifier** is the person who must verify the bug once it's fixed.
- >**Component** is the area where the bug was found.
- >**Found on** is the environment where bug was found.
- >**Version** is the version of the software where the bug was found.
- >**Build** is the build number of the software where the bug was found.
- >**DB** is the DB schema version of the software where the bug was found.
- >**Comments** are needed to provide additional info about the bug or about actions associated with the bug, etc.
- >**Severity** is about the bug's impact on the software.
- >**Priority** is about the bug's impact on the company's business.
- >**Also notify** can be used to add other individual(s) who should be notified about the fact of the bug filing and about changes to the bug.
- >**Change history** is needed to keep track of any change that happens to the bug.
- >**Type** is needed to specify the bug type: "Bug" or "Feature Request".
- >**Status** tells us whether the bug is open or closed. If the bug reemerges, we re-open it.
- >**Resolution** is needed to specify the stages of a bug's life.

8. Bug Resolutions (explanations are given with the assumption that bug report has Type "Bug"):

- >**Reported**: A bug was filed, but the developer to fix it has not been assigned.
- >**Assigned**: Assigned developer must start bug investigation.
- >**Fix in progress**: The developer is fixing the bug.
- >**Fixed**: The bug was fixed, but the bug fix hasn't been verified yet.

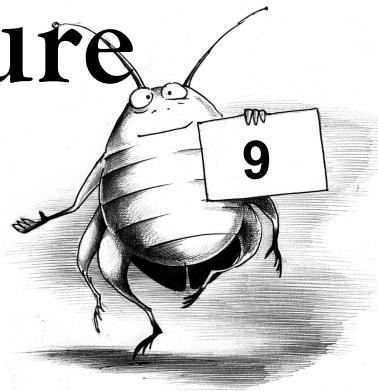
- >**Fix is verified:** The bug fix has been verified.
- >**Verification failed:** The bug fix verification failed; i.e., bug is reproducible after the bug fix.
- >**Cannot reproduce:** The developer cannot reproduce the bug.
- >**Duplicate:** The bug is a duplicate of another bug.
- >**Not a bug:** The bug is not considered to represent a problem (i.e., a deviation of actual from expected).
- >**3rd party bug:** The bug is in 3rd party software.
- >**No longer applicable:** The bug doesn't have any meaning anymore.

- Remember the principle of effective simplicity when you set up the BTS and/or introduce the BTP.

Questions & Exercises

- SL** 1. If you go to Test Portal>Application>Source code>shopping_cart.py, you'll see a section called "View Bugs" at the top of this page. Using the Training BTS, file all the bugs for shopping_cart.py.
2. Using the Training BTS log as many bugs as possible that relate to other areas of ShareLane. Pay special attention to the crispness of your Summaries and the comprehensiveness of your Descriptions. Note that all bugs from the Training BTS are removed at 00 minutes every hour.
3. Why do we need the BTS to file bugs?
4. Bring an example of a format for a bug Description.
5. List elements of the Web page.
6. How can you do a screen shot?
7. Explain the 2 parts of bug fix regression. Does it make sense to do the second part if the first part has failed (i.e., if the bug is reproducible)?
8. Why must the tester check the application version before verifying the bug fix?
9. What is the difference between bug Severity and bug Priority?
10. Why sometimes do we have to re-open bugs?
11. List all of the bug Attributes.
12. List all of the bug Resolutions.
13. What can testers do to prevent receiving bugs with a "Cannot reproduce" Resolution?
14. Draw a flowchart of the BTP.

Lecture



Test Execution: New Feature Testing

Lecture 9. Test Execution: New Feature Testing.....	258
Quick Intro.....	259
Test Estimates.....	261
Entry/Exit Criteria.....	263
Test Plan.....	264
Aggressive Testing From Jason Fisher.....	268
Lecture Recap.....	269
Questions & Exercises.....	270

**Be suspicious of anything that works perfectly –
it's probably because two errors are canceling each other out.**
Dave Bartley

**A computer lets you make more mistakes faster
than any invention in human history -
with the possible exceptions of handguns and tequila.**
-Mitch Ratliffe

Quick Intro

Test execution consists of two parts:

1. New feature testing (NFT)
2. Regression testing (RT)

After the code for the coming release is ready for testing, the release engineer pushes it to test environment, where testers perform a smoke test (also called a sanity test or a confidence test) to see if the **main flows of the main functionalities** still work. In the case of ShareLane, we try to register once and we try to checkout once. That's it – nothing fancy like negative testing. Please note that **during a smoke test we concentrate on finding blocking issues**.

Example

The smoke test fails if we cannot register a new user or if checkout cannot be completed.

The smoke test does NOT fail if a user can be registered, but the email with the registration confirmation has the wrong content, or the checkout is working, but the wrong discount is given, depending on the number of books in the shopping cart.

So, if you see non-blocking bugs during the smoke test, take a quick note, finish the smoke test, and log those bugs. The idea of a smoke test is to check the **testability of the software**; i.e., whether the software is ready to be tested or not.

Depending on the complexity of the system, a smoke test usually takes 5 to 30 minutes.

Usually there is no need in formal documentation for a smoke test; testers just do a quick evaluation based on their experience with the Web site. If documentation is needed (e.g., your manager wants it), you can do it in form of a checklist.

Naturally, if a blocking bug is found during a smoke test, the bug should be immediately communicated personally (following logging it into the BTS), so the developers and release engineers can start figuring out what's wrong. Once the problem is fixed, the code is pushed to the test environment again, and another round of smoke tests is performed, and so on. Once smoke tests pass, the release engineer freezes the code and the testers start the NFT. As one of my friends says, "Testers wait for freshly written code like a pride of hungry lions waits for an antelope. Both want to tear something apart."

**BTW**

Here is what I recommend that you do right before, or at the beginning of, the NFT for a major release: call a QA team meeting where each tester does a quick introduction to the features that he or she will be testing in the coming release. "Quick" means up to 5 minutes per person. The benefit here is that each and every tester will have a general idea about what's coming. This improves each tester's overall understanding of the system and may prevent "Not a bug" situations.

For example, George, who tests functionality A, finds a bug in functionality B, tested by Anitha. But George doesn't know that it's actually not a bug, but a legitimate functional change for that release. If there had been a team meeting, there would have been a chance that George would learn about that change and, thus, wouldn't waste his and the developer's time.

During the NFT we execute our test cases, log bugs into the BTS, communicate bugs to the developers, and verify bug fixes. There is nothing much to it, but let me share several quick things that you'll find useful.

Test Estimates
Entry/Exit Criteria
Test Plan
Aggressive testing from Jason Fisher

Test Estimates

As a rule, software companies have a Release schedule. Release schedules can come in many forms, from:

- "Everyone at ShareLane knows that we do a release every Thursday"
- to
- Complex document with a list of new features, references to specs, start/end times for each stage of SDLC, etc., for several future releases.

Naturally, the QA schedule is linked to release schedule.

Let's assume that, according to QA schedule, we have:

- Two weeks (10 working days + 4 weekend days – which can also become working days) for test preps
- Two weeks for test execution

In other words, we have:

- Two weeks to write test cases and do other test preps (e.g., create a test automation helper)
- Two weeks for NFT and RT

The problem is that even if testers stop eating, sleeping, and watching YouTube videos, there is a limit of how much testing can be done. Thus, there is an eternal conflict between:

- The avalanche of new features that marketing and PMs want to push and
- The abilities of testers to do proper testing

To find the status quo between the desired and the possible, testers submit **test estimates**. A test estimate includes:

- Time for test preps
- Time for the NFT

After the spec is written, the test manager asks the tester to read it and evaluate how much time it will take to:

- Write test cases and do other preps for the NFT
- Execute the NFT

The tester reads the spec, talks to the PM and the programmer, and – based on all that info – he or she submits two figures to his manager: one figure is the estimated time for test preps and the other figure is an estimated time for the NFT. The time is usually represented in hours.

Example

A tester was asked to create a test estimate for Spec #8112 "New search features". The tester gave the following to his manager:

50 hours for test case generation + 20 hours to write an automation tool
60 hours for the NFT

If we have 2 weeks (80 hours) for Test preps, then the tester has 10 hours left (80-50-20). Those 10 hours can be spent helping other testers, or generating more test cases. It's also good to have some time reserved in case the estimate is not correct or in case something doesn't go as planned, e.g., if the PM has to change a spec.

In this example it is more complex with test execution. There are only 20 hours left (80-60) for RT. Will it be enough? It depends on number of factors. For example:

- How many test cases do we plan to execute for RT?
- What is the overall load on the testers? Maybe some testers have a lesser load and can help their amigos.

How do we write test estimates? The challenge here is that test estimates are created after the spec was simply **read**. But the difference between spec reading and the actual testing of the spec's feature is like the difference between reading about skydiving and actually jumping from the plane: **HUGE**. Therefore, when you are writing your test cases, or executing those test cases, hundreds of nuances come to light... nuances that you couldn't even imagine during spec reading.

Below are the factors I recommend that you consider while writing test estimates:

1. What is the complexity of the feature to be tested?
2. Do you have any experience with testing similar features?
3. Is there any planned integration with the vendor's software?

1. What is the complexity of the feature to be tested?

The rule is: The more complex the feature, the more nuances will emerge; the more nuances that emerge, the more time needs to be spent on testing.

2. Do you have any experience with testing similar features?

For example, if you are experienced with testing Checkout, you'll spend much less time testing the addition of another type of credit card compared to a person who has never tested Checkout.

An **experienced** person does a lot of things automatically; he or she knows where to look for answers and who can help.

An **inexperienced** person MUST spend time exploring things, reading documentation (which often sucks or is outdated), bothering various people with questions, etc.

The bottom line is: If you are inexperienced with some feature (especially a complex one) make sure to build your learning curve into your test estimates.

BTW

In the case of complex software, there are people in the QA department who become experts in particular parts of that software. This is a great thing, but there is a danger that an expert can become ill or even depart this wonderful world right in the middle of something important – for example, during test case generation. So a QA manager must make sure that the testing of complex features is well documented and that this documentation is properly maintained. Also, it's a good idea to assign 2 testers (at least) to each complex area – this way we have some "insurance" in case of an unfortunate incident with one of them (unless of course they get into the accident together).

3. Is there any planned integration with the vendor's software?

Let's say that we integrate our software with a payment processor. The test architecture looks like this: Our test environment "talks" to the vendor's test environment. In case there is a bug (or another type of problem, e.g., a server is down) on our side, we can solve the problem like this: "Hey, Billy, can you sort this out?" But in the case of a problem on the vendor's side, it can take hours or even days before they might fix it; our PjM must talk to the vendor's PjM, the vendor's PjM must talk to their developers; their developers might be busy with other stuff, etc. Therefore, the gist of the problem is that our issues are not their issues. Thus, if there is a planned integration with the vendor's software, you should always increase your test estimate for the NFT. How much should the increase be? Each situation is unique; the complexity of software and of the integration, and any previous experience with the same vendor are important factors here.

BTW

After you have your final test estimate, increase it 10%, just in case of any unforeseen circumstances.

BTW

A very simple and working approach that I've seen is to dedicate 1 day of test preps + 1 day of NFT for each page of the spec.

Entry/Exit Criteria

Entry criteria are certain conditions that allow you to start something. For example, to make a phone call you have to have a working phone, a connection, and the phone number of the recipient. Therefore, we can say that entry criteria for a phone call includes 3 conditions:

- A working phone is available.
- A connection is available.
- A phone number is known.

Exit criteria are certain conditions that allow you to declare that something is finished. For example, lunch at the restaurant is finished when the bill is paid. So, the exit criteria for a meal at a restaurant is:

- The bill is paid.

Both Test preps and Test execution have their own entry/exit criteria. See the examples below. Of course, every company has its own rules, so just grasp the concept.

Example

Entry criteria for Test preps: All specs are frozen.

Exit criteria for Test preps: All test preps (test cases, test tools, etc) are completed.

Example

Entry criteria for Test execution: the code is frozen; the test cases for the NFT are ready to be executed.

Exit criteria for Test execution: NFT and RT are done; there are no open P1 and P2 bugs.

BTW, the main purpose of the Go/No-Go criteria document is to specify entry criteria for a major release.

Of course, the best way to keep things clean and transparent is to have entry/exit criteria for all necessary situations well defined and documented.

Test Plan

The test plan is the master document about the activities regarding the testing of a certain feature (or another possible subject of testing, e.g., how the system handles a load).

Let's forget about software for a minute.

Below are two situations:

1. You have to drive to the nearest supermarket to buy some milk.
2. You have a wedding coming soon.

Does it make sense to sit and write down a plan...

- ...in the first case? No, it's absurd. Just go out and get the milk.
- ...in the second case? Absolutely, because a wedding involves a large number of activities and people, and it has a significant meaning. (Of course, I'm talking about a traditional wedding, not a "let's get married NOW" thing in Las Vegas.)

The two situations above are different from a planning perspective because they provide different answers to the following questions:

- How quickly must the project be completed?
- How many people and activities are involved in the project?
- What is the significance of the project?

Let's get back to our start-up business. Consider the following:

- New features in start-ups are usually developed (and tested) very quickly (taking just days or weeks).
- There is usually one PM, one developer, and one tester per each feature.
- Any feature can be changed, removed, or deprecated overnight.

So, ***as a rule, in the start-up environment there is no need to create a test plan.***

BTW

Sometimes a tester is asked to write a test plan by his or her manager when the manager doesn't know what a test plan is. If you have a manager like this, ask him or her: "Do you want me to create a formal test plan, or do you want me to generate test cases?"

To be fair, there are situations when a separate test plan *is* needed; e.g., for long-term projects, or for projects that include integration with a vendor's software.

Test plans come in many shapes and forms, but the majority of them have their roots in the test plan document introduced by ANSI/IEEE Standard 829-1983. For my projects I use a simplified version of this. You can find it in Downloads on QATutor.com. Let's look at the main sections.

General info

Introduction

Schedule

Feature documentation

Test documentation

Things to be tested

Things not to be tested

Entry/Exit criteria

Suspension/Resumption criteria

Other things

GENERAL INFO

Author	<Name (-s) of test plan author (-s)>
Version	<Version of test plan, e.g., 1.0>
Last updated	<Date and time of ANY last change inside this document>
Status (Draft/Finished)	<"Draft" is while writing the test plan. "Finished" is after the test plan is finished and ready to be used. >
To-do list	<Things to do to finish the test plan. This section should be blank when the status is "Finished".>

INTRODUCTION

This is a short intro to the feature that is going to be tested*.

**For simplicity, I assume that we are testing a certain feature, but you should remember that we can also test other things – site performance, for example.*

SCHEDULE

Here you specify detailed info about things that should happen, when they should happen, and who is responsible.

Example:

Date	Deliverable	Responsible person	Team
09/12/08	Coding is finished	Billy	Dev
09/12/08	Test cases are finished	George	QA
09/26/08	NFT and RT are finished	George	QA
10/01/08	Go/No Go	Linda, Billy, George	PM, Dev, QA

FEATURE DOCUMENTATION

Here you list and provide links to all relevant documents that specify how a feature should work. Usually those documents are specs.

Example:

Title	Notes
Spec #1288 "Redesign and Rearchitecture of Checkout	Link
"Rules on credit card transactions in North America" by Visa Corp.	Link

TEST DOCUMENTATION

Here you list provide links to the test documentation needed to test this feature. Usually these documents are test suites.

Example:

Title	Notes
Test Suite #4837 "Redesign and Reachitecture of Checkout	<u>Link</u>

THINGS TO BE TESTED

Here you describe what you are going to test and how.

Example:

Subject of testing	Testing approach	Needs automation helper? (leave blank for "no")
Checkout with Visa	1. Component testing: a. Positive testing: card valid + enough balance b. Negative testing: card is invalid/not enough balance. 2. Integration testing: a. Integration with Shopping Cart 3. System testing: Search->View book->Shopping cart->Checkout	Credit Card Generator. This tool has already been written.
Checkout with MasterCard	see approach for Visa	see approach for Visa
Checkout with AmEx	see approach for Visa	see approach for Visa

THINGS NOT TO BE TESTED

Here you specify which things are not going to be tested.

Example:

Subject of testing	Why we don't test it
All features other than	We are going to test Checkout only. Other features will

"Checkout"	be tested separately during NFT and/or RT.
Load/Performance/Reliability and security	For this release we'll focus on feature testing only.

ENTRY/EXIT CRITERIA

Here we specify the entry/exit criteria for the NFT of the feature.

Example:

Entry Criteria	Exit Criteria
Code is frozen; test cases for NFT are ready to be executed	NFT and RT are done; no open P1 and P2 bugs

SUSPENSION/RESUMPTION CRITERIA

Suspension criteria are certain conditions upon which testing must be suspended.

Resumption criteria are certain conditions upon which testing must be resumed after suspension.

Example:

Suspension Criteria	Resumption Criteria
7 or more open P1 bugs	6 or less open P1 bugs
Testing is blocked	Testing is unblocked

OTHER THINGS

Here you specify whatever else you want included in your test plan (e.g., training, hardware/software requirements, contact info, sign-off procedure, etc.)

Aggressive Testing From Jason Fisher

This testing technique was introduced by my dear friend, top Ruby developer Jason Fisher.

In start-ups we have loads of small new features to release, and releases might happen several times a week. There can also be situations when there are many developers and only one tester. Here is what can be done in such an environment:

1. After the coding is finished and the code is pushed to the test environment, the developer and the tester get together and begin discussing test scenarios. They outline major flows on a white board, and they decide who tests what. Right – both the developer and the tester will be doing testing.
2. The testing starts, and the communication is done person-to-person or via Yahoo! Messenger. All bugs are fixed by the developer as soon as they are found. New builds are pushed promptly after the bug fixes are checked in into the CVS.

3. After the major flows seem to work, the developer gets back to his or her business and the tester spends several hours:

- Planning the testing, e.g., using the black list-white list technique
- Executing the testing, e.g., by going through the flows on the white list

4. After found bugs are fixed and verified, the tester does a quick system test followed by an acceptance test. If everything is fine, the tester gives the green light to push to production.

It sounds simple, but it works fabulously.

Lecture Recap

1. The NFT starts after the smoke test passes.

2. The purpose of a smoke test is to check the testability of the software. In other words, we are looking for bugs that block testing.

3. During the smoke test we check the **main flows of the main functionalities**.

4. There is usually no need to create test documentation for a smoke test.

5. It's a good practice to call a QA meeting (right before or at the beginning of the NFT for a major release), where each tester gives a short presentation about the features that he or she is going to test for the coming release. This improves each tester's overall understanding of the system and may prevent "Not a bug" situations.

6. Test estimates are needed to find a balance between the avalanches of new features that marketing and the PM want to introduce and the ability of testers to perform proper testing.

7. Test estimates consist of two parts: a time estimate for test preps and a time estimate for the NFT.

8. Things that can affect test estimates:

- What is the complexity of the feature to be tested?
- Do you have any experience in testing similar features?
- Is there a planned integration with the vendor's software?

9. Entry criteria are certain conditions that allow to begin something.

10. Exit criteria are certain conditions that allow something to be considered finished.

11. The test plan is a master document about the activities for testing a certain feature (or another possible subject of testing; e.g., site performance).

12. Here is the simplest structure of a test plan:

General info
Introduction
Schedule
Feature documentation
Test documentation
Things to be tested
Things not to be tested
Entry/Exit criteria
Suspension/Resumption criteria
Other things

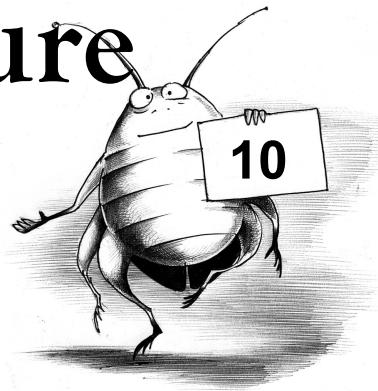
13. As a rule, start-up testers don't need to create test plans: features come and go too quickly, teams are small, and often there is not even enough time for the essential things like test case generation.

14. The core idea of aggressive testing developed by Jason Fisher is to find and fix important issues ASAP.

Questions & Exercises

1. Does a P1 bug found during a smoke test automatically mean that the smoke test has failed?
2. What is the BTS Severity of a blocking bug?
3. Why do we need a smoke test before the NFT?
4. Why do we need test estimates?
5. Think of some examples of Entry/Exit criteria from real life.
6. As a rule, why don't we need test plans in start-ups?
7. In which cases does it make sense to have a test plan?
8. Define Suspension/Resumptions criteria.
9. Describe the Jason Fisher's aggressive testing technique.

Lecture



Test Execution: Regression Testing

Lecture 10. Test Execution: Regression Testing.....	272
Quick Intro.....	273
How To Choose Test Suites For Regression Testing.....	273
How To Resolve The Contradiction Between Our Limited Resources And The Ever-Growing Number Of Test Suites.....	278
Automation Of Regression Testing: Do It Right OR Forget About It.....	278
When Regression Testing Stops.....	296
Lecture Recap.....	297
Questions & Exercises.....	299

**It is the greatest of all mistakes to do nothing
because you can only do a little.
Do what you can.
-Sydney Smith**

**While we are free to choose our actions,
we are not free to choose the consequences of our actions.
- Stephen R. Covey**

Quick Intro

Regression testing (RT) is the second part of Test execution. The gist of RT is checking whether any changes made in the software for a new release have broken any features that existed prior to those changes.

Let's assume:

- We have 5 test suites for NFT and 21 test suites for RT
- There is no way that we can execute all of the 21 test suites within the time frame given for RT.

This situation brings up two questions:

1. How can we choose several test suites out of those 21 to do proper RT?
2. What is the solution for RT in the long run? We'll have 26 test suites for RT for the next release, and so on. Think about the snowball effect.

Thus, we have two topics here:

1. How to choose test suites for RT
2. How to resolve the contradiction between limited resources (people and time) for RT and the ever-growing number of test suites

How to Choose Test Suites for Regression Testing

The first question is this: What parts of the software can be broken?

On the one hand:

- a. It's often extremely difficult for a programmer to know how a change in one part of the software will echo in other parts of the software.

AND what's even worse:

- b. Programmers sometimes change the software **without even trying** to figure out if their changes might break something.

Let's illustrate item b.

Example

Let's assume that a programmer writes some new functionality for the shopping cart. In the course of programming, he or she decides to drop (i.e., remove) a certain column of a certain DB table, because in that programmer's opinion it seems to be useless. A problem might occur if he or she doesn't check to see if that column is still used by some other part of software – e.g., by register.py. The important thing to understand here is this: that other part of the software can be completely unrelated to the "Shopping cart" feature, but it can still be broken in the course of programming the shopping cart.

Another popular scenario is this: A programmer changes a certain function in the example_one.py file without realizing that the function is imported into and used by the example_two.py file. So, the functionality delivered by the example_two.py file can be completely or partially broken because of the change in the example_one.py file.

On the other hand, testers usually know (or can make an educated guess) which features will be directly affected by a change.

Example

If the programmer changes how the shopping cart works, we know that the "Shopping cart" feature is being affected.

Here is my approach is to single out 3 groups of test suites for RT.

The 1st group must contain test suites that check legacy (i.e., existing) features that are **directly affected** by a change in software.

Rationale: A programmer is most likely to mess up the code of the feature that he or she is changing.

Example

If the programmer changes how the shopping cart works, we know that the "Shopping cart" feature will be affected. So, for the **1st group** we shall use the test suite dedicated to testing the shopping cart.

The 2nd group must contain test suites that check legacy features that may somehow **depend** on the changed features.

Rationale: While changing feature A, a developer can break a piece of code that feature B depends on.

Example

The Checkout functionality depends on the Shopping cart functionality. When a user presses the "Proceed to Checkout" button, 3 key-value pairs are passed on to checkout.py.

Example of 3 key-value pairs:

```
book_id=5
q=1
total=1070
```

Example of a URL containing those key-value pairs:

http://www.sharelane.com/cgi-bin/checkout.py?book_id=5&q=1&total=1070

SL Try it yourself by adding book "The Power of Positive Thinking" to the shopping cart and pressing button "Proceed to Checkout".

If the programmer breaks the code inside `shopping_cart.py` so that

- the key and/or key's value is not generated at all (e.g., "book_id=" instead of "book_id=5")
- the incorrect key name is generated (e.g., "id" instead of "book_id")
- the incorrect key's value is generated (e.g., "book_id=4" while book_id must (in our example) equal 5)
- the Web form cannot be submitted at all or is submitted to the wrong application core file (e.g., "register.py" instead of "checkout.py")

then the checkout functionality will be broken. Thus, if the Shopping cart functionality was changed, for the **2nd group** of RT we must use test cases that test Checkout.

There is also a **3rd group** that we'll discuss in a minute.

Look at the table below where you can see an example of the **1st group** and the **2nd group** with the associated test suites.

Group	Test Suite ID
1 st	TS1111
	TS2222
	TS3333
2nd	TS4444
	TS5555
	TS6666
	TS7777
	TS8888

Executing each of those test suites takes time. Now let me ask you this: How do we know if we have enough time to execute those test suites within the time frame given for RT?

Let's assume that we have only 1 tester (George) and 2 weeks for RT. If George works 8 hours a day, then 2 weeks equals 80 official business hours + 32 hours (if he works both weekends). It also can convert into 336 hours (24 x 14) if he is willing to forget about everything else but RT.

Question: Can George execute all 8 test suites within 80 hours?

Answer: To answer, we should know how much time the execution of each test suite takes.

Question: How can we know that?

Answer: Each company computes this in its own way. Some companies have special software to track the time required for test suite (or sometimes even test case) execution; in other companies, the tester who did the last execution puts the amount of time spent somewhere in the header of the test suite. In any case, it's very important to know **approximately** how much time it takes to execute each test suite. I say "approximately" because that time can vary depending on a number of factors – for example, if that tester has had prior experience executing that particular test suite or not.

Let's assume that we know approximately those execution times.

Group	Test Suite ID	Time for Execution (hours)
1st	TS1111	10
	TS2222	15
	TS3333	17
2nd	TS4444	18
	TS5555	12
	TS6666	14
	TS7777	26
	TS8888	19
Total		131

Here, even if George worked both weekends (112 hours total), he would still need 19 (131-112) hours to finish the test suite execution. Those 19 hours can be distributed over 14 days. He'll need to work approximately 1 hour and 20 minutes ($19 \times 60 / 14$) on top of 8 hours a day for 2 weeks. This is how things often work in start-ups.

BTW

131 hours of work over two weeks doesn't hurt at all if you work as consultant: you'll make \$4585 if your hourly rate is \$35.

If you work as a permanent employee, you'll get nothing for your overtime beyond your usual compensation. So make sure to work for a promising start-up where your sacrifice will be compensated when your stock options are converted into shares of a successful company.

We'll talk about this soon.

Let's assume that we are humanitarians, and we don't want George to work more than 40 hours a week. In that case, there are 51 hours (131-80) that we need but don't have. What can we do? Among other things – like outsourcing testing to other companies or borrowing folks from marketing – we can use **test suite priority** to single out the most important test suites.

Group	Test Suite ID	Time for Execution (hours)	Test Suite Priority
1st	TS1111	10	1
	TS2222	15	3
	TS3333	17	4
2nd	TS4444	18	4
	TS5555	12	2
	TS6666	14	1
	TS7777	26	3
	TS8888	19	2

If we execute test suites with

Priority 1 – we'll need 24 hours (10+14)

Priority 1 and 2 – we'll need 55 hours (24+12+19)

Priority 1, 2 and 3 – we'll need 96 hours (55+15+26) – and that doesn't work for us because it's more than 80 hours.

Therefore, we can execute Priority 1 and 2 test suites (55 hours), and we can spend the remaining 25 hours to execute:

- P1 and P2 test cases from those test suites with Priority 3 and 4 and/or
- Test suites from the **3rd group**

Now, let's talk about the **3rd group**. As a rule, the majority of test suites don't fall into the **1st** and **2nd** groups. But that majority surely needs to be tested too, because, as we know, a change in one part of the software can often break other seemingly unrelated part(s) of the software. It's usually out of question to execute ALL of those test suites during RT for a single release, so what we can do is to set up a plan to execute all the test suites within a defined period of time – e.g., 6 months.

Example

If we have 21 test suites for RT, and we do a major release once a month, we can decide to execute 4 test suites for the first 5 releases and 1 remaining test suite for 6th release. This way, we'll execute each test suite at least once every 6 months.

Please note that there are nuances here – e.g., test suites that we plan for the 3rd group can fall into the 1st or 2nd group; there are also situations where we have to execute the same test suite more than once for RT during those 6 months, etc.

How To Resolve The Contradiction Between Our Limited Resources And The Ever-Growing Number Of Test Suites

We can resolve this contradiction by the following means:

1. Prioritization of test suites and test cases
2. Test suite optimization
3. New hires or outsourcing
4. Test automation

1. Prioritization of test suites and test cases

We've already discussed this topic. The prioritization of test suites and test cases is the cheapest and, in many cases, most effective solution to resolve the issues of contradiction. If you use test suite/case priorities as the criteria for test suite/case selection for RT, you should keep those priorities up-to-date. The test suite/case priority must be changed, for example, when a tested feature loses its importance or, on the contrary, becomes more important.

2. Test suite optimization

Many old test cases can be optimized in two ways:

- a. Reduction in the number of test cases
and/or
- b. Simplification of the test case execution

For point a – Very often it makes sense to review test cases that were written some time ago to see if they make any sense today; for example, some features can be deprecated, or maybe the test case is a duplicate of another test case. In some cases, it makes sense to retire whole test suites.

For point b – Some test cases can be simplified because new helper tools are written.

3. New hires or outsourcing

In some cases, this makes sense; in some cases, it doesn't. To hire new testers or to outsource testing is the most linear way to resolve the contradiction. I know of both success stories and failure stories. There is no universal advice here; it all depends on the company and its type of business.

4. Test automation

This is one of the most complex and controversial topics, so I have created a separate section for it. Read on.

Automation Of Regression Testing: Do It Right OR Forget About It

In general, test automation includes a myriad of different tools and techniques for a myriad of different purposes: code analysis, link checking, load/performance testing, code coverage, unit testing... this list goes on and on. There are tons of books written on this subject.

Today, we'll take a look at one specific kind of test automation: automation of regression testing. Let's call it test automation or "**TA**".

My purpose here is to:

- Warn you about the dangers of incorrectly done test automation
- Share some thoughts on how to do test automation right

Please note that we'll be talking about test automation **in the start-up environment**.

Here is a typical Silicon Valley story:

A Story about the Merciless Automator, Benny M.

In our beloved ShareLane.com we've accumulated 78 test suites. It hurts to think about it, but somebody HAS to execute them. It hurts even more to know that after each major release, the number of test suites keeps growing.

Somebody comes up with an idea to hire a test automation engineer. In a couple of weeks, this test automation engineer is hired. His name is Benny M.

The QA manager calls a special meeting, during which Benny is presented very much like the savior of humanity. When he is asked to speak, Benny gives a simple yet powerful one-liner: "**I'm going to automate it all!**" How in the world can somebody not fall in love with a person like that!

There is a little problem, though: the company will have to spend thousands of dollars to buy a special automation program. However, we are so exhausted by manual regression testing that we are willing to spend any kind of money to relieve ourselves of this hassle! Anyway, we decide to buy a special program called SilkTest (the original creator was Segue; as of September 2008, the owner is Borland) so Mr. Benny can feel comfortable - SilkTest is his tool of choice, you know.

After one month, Benny makes a presentation. And, my friends, what a presentation it is! He pushes a button, and in some magical way the SilkTest agent opens a Web browser window, enters a URL in the address bar, and clicks "Go". Next, on the Web page it types an email and a password, clicks "Log in", makes a purchase – and at the end, it compares the expected and actual results!

Witnesses of the miracle give Benny a standing ovation. Benny nods like a person who humbly accepts well deserved admiration and retires to his quarters, while we, mere mortals, gather at the cappuccino machine to make our prediction: it looks like in a couple of months we'll have all our 78 test suites automated! No more sleepless nights – just press the button and go to the bar. One word: **MAGIC!**

But when we enter the regression stage of test execution and ask Mr. M. to run his automation, he says that the UI has changed and so his automation needs to be updated too. *For example, when Benny was writing his test script for the registration flow, we had a button for log in called "Log In", but the developers made some changes and that button is now named "Log On", so the automation scripts must be updated to comply with that change.*

"Come on!" says Benny. "I'll fix it in no time." He spends another two weeks making fixes, and on the day before the Go/No-Go meeting he finally runs 2 out of 10 automated test suites. With the next release, we have the same story, and the QA manager decides to fire Benny and hire two black box testers who will be much cheaper and much more effective. SilkTest stays on Benny's abandoned machine as a silent reminder about wasted money and gained wisdom.

The inconvenient truth is that if a software company ends up wasting \$100K (the cost of a commercial automation tool plus the services of an automation engineer) to gain wisdom about test automation, it's a cheap price. There are many cases when millions of dollars are flushed down the toilet until management realizes that something really isn't working here.

What exactly was the problem with ShareLane TA?

Was it SilkTest? No.

SilkTest is just a tool. Each tool is effective as long as it's **properly applied for a certain purpose under certain conditions**. If you use a hammer instead of a screwdriver to open the back of your Rolex, it's not the hammer's problem if your Rolex is smashed to pieces.

BTW, we'll talk about SilkTest and other similar tools in a minute.

Was it Benny? Yes and no.

Benny is definitely an asshole. For example, he should never have promised to automate 100% of the regression tests, and he should never have attempted to automate flows that are prone to frequent modifications. On the other hand, Benny was a good programmer, but we didn't give the correct direction to his boiling energy. More precisely, we didn't give him any direction at all.

So...

Was it us? Yes.

Once we made a decision that test automation was necessary, we had to develop the **correct approach towards it**. We didn't do this. Instead, we chose the easy way: **hire a dude who would come and solve our problems**. That didn't work out, and we can only blame ourselves. But during the process of blaming, we can learn a lesson or two to help us avoid the same mistakes in the future.

Let me share with you some thoughts and examples regarding TA that I've learned. Please pay attention.

TA of regression testing usually comes in 2 forms:

1. Helper tools (let's call them "helpers")
2. Programs for automated regression testing (let's call them "automation scripts")

The **purpose of helpers** is to:

- Automate concrete repetitive manual tasks (Account Creator)
- OR
- Give testers the essential means to perform testing (Credit Card Generator)

The **purpose of automation scripts** is to execute regression test cases and report the results.

On the high level, test automation consists of 3 components:

Component 1: **A certain task** – e.g., *execute test case #1 and report results*

Component 2: **A certain automation program used to automate that task** – e.g., *Benny's SilkTest script that executes test case #1 and reports results*

Component 3: **A certain piece of software tested with the help of or by an automation program** – e.g., *the checkout.py file is tested by Benny's script that executes test case #1 and reports results*

The question, "What to automate?" is equivalent to the question, "What task shall we automate?"

For example, we execute test case #1 manually from release to release. How can we determine **whether or not it makes sense to automate** its execution?

My friends, I'm here to tell you that **in the majority of cases, it's harder to select a good candidate for automation (in other words, some task worthy to be automated) than to perform the automation itself**. You can pick up Python skills within days, but it takes years of experience to be able to predict whether automating a certain task will result in saving or wasting a company's money. We at ShareLane invited a fiasco with TA because we didn't ask Benny **how he would select candidates for automation**, or, in other words, **what criteria he would use to single out test cases that were worthy to be automated**.

The most important criterion for TA is the **stability (maturity) of the software to be tested** (Component 3). That criteria should be applied every time when we evaluate some part of our application for automation eligibility.

Take a quick look at the test case below (**don't execute it**):

1. Go to <http://main.sharelane.com>.
2. Click link "Test Portal".
3. Click link "Account Creator".
4. Press button "Create new user account".
5. Copy email to the clipboard.
6. Go to <http://main.sharelane.com>.
7. Paste user email into textbox "Email".
8. Enter "1111" into textbox "Password".
9. Press button "Login".
10. Enter "expectations" into textbox "Search".
11. Press button "Search".
12. Press button "Add to Cart".
13. Click link "Test Portal".
14. Click link "Credit Card Generator".
15. Select "Visa" from drop-down menu.
16. Press button "Generate Credit Card".
17. Copy card number to the clipboard.
18. Go to <http://main.sharelane.com>.
19. Click link "Shopping cart".
20. Press button "Proceed to Checkout".
21. Select "Visa" from drop down menu "Card Type".
22. Paste card number into text box "Card Number".
23. Press button "Make Payment".
24. Write down order id: ____.
25. Click link "Test Portal".
26. Click link "DB Connect Utility".
27. Make database query:
select result from cc_transactions where id = <order id>;

Expected result: "10"

As you remember, there were several problems with this, one of which was that the maintainability of that test case might become a headache. For example, we have to modify the test case if the "Make Payment" button is renamed "Order". Therefore, we made it modular by doing several things – one of which was to appeal to the common sense of the test case executor; i.e., we assumed that a human could FIGURE OUT what to do if we just wrote "Login" as an instruction.

Problem number 1: Whatever tool we use to write TA, we cannot rely on its common sense, because the program only runs if you provide it with concrete, precise instructions. Therefore, every time when instruction must be changed (e.g., button "Log in" is renamed into "Log on"), TA must be changed too. So, we return to our original problem: maintainability, but in this case, it's **maintainability of TA**.

Problem number 2: It's a hassle to modify the steps inside a test case, but it's much more painful and time consuming to modify/rewrite TA! When you modify your test case, you just retype the steps inside that test case, but when you modify your TA, you have to do the programming!

Thus, **maintainability is the number one issue when we create TA**. But why do we have an issue of maintainability in the first place? Simple: because the software to be tested was subject to changes. **Change means maintenance; no change means no maintenance.**

So, it really is simple: **the best test case to be automated is a test case that tests software that is not subject to change.**

Problem number 3: In a start-up environment, the modifications made to software are frequent and dramatic – in many cases, we might simply decide to totally redesign the UI or even replace our old product (e.g., photo sharing) with an absolutely different one (e.g., video sharing).

Test engineers cannot stop software development just to make that software more suitable for test automation. What are the chances for further employment of the test automation engineer who approaches a product manager and says, "You know, all your modifications really screw up my test automation. Can't you just calm down with your creativity and let me do my job?"

Frequent modifications to software are INEVITABLE in a start-up environment. This means that frequent maintenance of the TA that tests that software is also INEVITABLE.

BTW

Here is a good analogy about the relationship between software maturity and TA: Parents usually don't buy \$200 shoes for 3-year-old children. Why? Because a 3-year-old's feet will grow in several months and that \$200 will be a waste of money.

By way of analogy, it doesn't make sense to invest seriously in TA for "young," changing software.

Now, after you have an idea about the **TA killer** called MAINTENANCE, let's have deeper look into the common types of TA:

1. Helpers

2. Automation scripts:

- a. Tools for component automation
- b. Scripts for end-to-end automation

1. HELPERS are tools that assist the manual execution of test cases. Reading the previous chapters, you've had many chances to use ShareLane helpers – e.g., Account Creator and Credit Card Generator. No wonder helpers are loved by everyone! As far as I've seen in the majority of cases, helpers are the most effective, useful, and economical type of TA.

Helpers are usually the first pieces of TA. Why? Because manual testers usually have a very good idea about small things that can make their life easier, so if there is a person who can help with TA, it's natural that he or she is asked to automate those little things. Let's look at account creation. There are

special test cases that we use to test account creation flow, but **in all other cases, we just need a new account**. So why not automate the **repetitive, boring task** of generating a new account? There are also situations where testing is not possible without helpers – e.g., when we need to use test credit cards.

But why are helpers the cheapest type of TA? Mainly, because there is usually no – or very little – maintenance required. The reason is simple – **helpers don't test software**. Therefore, if the software changes, the helpers don't necessarily need to be changed.

Example

If a programmer completely rewrites the checkout flow, the helper used to generate credit cards doesn't need to be changed, because whatever the checkout flow is, the credit card attributes – e.g., 16 digits in number – stay the same!

Of course, there ARE cases when the helper's code needs some maintenance.

Example

Sometimes we have to change the helper's code – for instance, when there is an expansion of the tested software. For example, we must update the Credit Card Generator if we plan to accept another credit card (e.g., Discover).

In my experience, even if a helper needs to be changed, it doesn't usually require much time and effort to do it. The trick is to create helpers **for appropriate tasks and in correct ways** (we'll talk about this a minute).

BTW

There are also cases when the helper's code relies on the tested software's code – e.g., Account Creator uses lib.py (create_account.py imports qalib.py that imports lib.py). In general, I don't recommend doing it, because TA engineer will not have a full control over that helper and that helper usually requires more maintenance compared to helper that doesn't depend on tested software.

Once again, **in the majority of cases helpers are the most effective, useful, and economical type of TA**. My TA engineer career started many years ago with helpers, and many of them are still saving time and effort for my fellow testers.

Let me give you some simple arithmetic about helpers. Below is the initial data regarding Account Creator.

- It took me about 30 minutes to write and test Account Creator.
- It takes 4 seconds to create a new user account and do an auto log in with Account Creator.
- It takes about 1 minute to create a new user account and log in manually.

Let's assume that:

- ShareLane has 10 testers

- On average, each tester creates 20 new accounts daily

This means that 52,000 minutes ($20 \times 10 \times 260$ working days) or 866 hours are needed for account creation annually. With automation it takes only 1/15th of that time – i.e., 58 hours ($866/15$) – and note that this requires only a TINY EFFORT to create an account with Account Creator! Therefore, the delta is 808 hours (866-58). If testers make \$45 an hour on average, that's \$36,360 in savings per annum. So – were those 30 minutes well spent?

BTW

Some helpers automate tasks that require hours of hard, concentrated, manual labor and/or special expertise – e.g., creating financial files for an interbank exchange.

BTW

Another benefit of using helpers is that boring tasks like the repetitive creation of a new user account can exhaust a person faster than a creative task like generating a new test case. Helpers not only save time, but they also enable manual testers to be less tired and thus more productive.

As you can see, a **TA engineer who writes helpers can save millions of dollars for his or her company**. I think that monetary savings is the primary reason for TA in the first place, right?

2. AUTOMATION SCRIPTS.

a. *Tools for component automation*

SL One example of component automation is a script that performs an isolated testing of the discount rate depending on the number of books inside the shopping cart (Test Portal>Automation Scripts>Test Discounts).

Previously we came up with a set of tests for discount rates using equivalent classes. Equivalent classes (EC) is a good, but not 100% reliable, technique, because the code can contain some weird bug that won't necessarily be caught by the EC approach.

SL With component TA for discount rates we can generate thousands of inputs, **directly** apply them to the function `get_discount()` (Test Portal>Application>Source Code>`shopping_cart.py`), and get more reliable results than in case of manual testing.

The *Test Discounts* tool implements a **short, isolated** test flow – we just keep feeding various inputs to the `get_discounts()` function.

The words "short" and "isolated" are major points when we talk about component automation because:

- If the test flow is short, there is less probability that changes to software will get in the way of test execution.

Think about two roads with the same traffic intensity: the length of the first road is 50 miles; the length of the second road is 1 mile. As a rule, roadwork is less likely to be performed on the 1-mile road than on the road that is 50 miles long. Many factors can determine roadwork, but in general a shorter road means a lesser possibility of it.

- *If the test flow is isolated, then changes to other parts of the software will not affect the tested component.*

Brain Positioning

If one piece of software changes, another piece of software integrated with it can be seriously affected by that change.

SL For example, if we remove the get_html() (Test Portal>Application>Source Code>lib.py) function, ShareLane will stop functioning, because each of the scripts inside the application core uses that function to generate the html code for the page!

Can the attractiveness of the TA component be affected by MAINTENANCE? Of course! But if we find a good candidate for the TA component, then we'll save plenty of money for our company and, as with the Test Discounts tool, provide much better coverage.

SL It took me about an hour to create test_discounts.py (Test Portal>Automation Scripts>Test Automation Source Code>test_discounts.py), and it takes 4 seconds to test the discounts now, so if we test the discounts once a week and it takes 15 minutes to run manual tests, then my efforts will be justified in a month. Also note that test_discounts.py provides comprehensive coverage for this very important feature (any feature dealing with a user's money is important!).

Brain Positioning

Even if a TA engineer selects good components to write TA and creates well designed TA, there is still the risk of maintenance.

Here is the logic: if you have 25 roads where the length of each road equals 1 mile, there's a greater probability of roadwork when compared to a single 1-mile road. So, the more components you automate, the higher the risk of maintenance. What is the logical consequence of extensive TA? A situation known as: "I come to work to maintain TA."

So think twice before you start automating something. **It's better to have 10 robust pieces of TA and have time to write new TA, do black/grey box testing, and maintain the existing TA than to have 30 pieces of TA and spend all week on maintenance.**

b. Scripts for end-to-end automation (E2E TA)

This is the most desired (by those who have NO IDEA) and most dangerous kind of TA. Stories about wasted money usually happen when folks in power who know nothing about the subject adopt the motto "Let's automate it all" to implement TA projects.

What is the major problem with E2E TA? The length of the road ... remember our example about the two roads. E2E is about the road that is 50 miles long, and hence there is much more chance for roadwork compared to the 1-mile road. We know that software in start-ups changes rapidly, so:

- If you test one component, you are taking a risk, but that risk can be justified if the component is chosen correctly and the automation is implemented correctly.

BUT

- If your test flow goes through several integrated components, your risk increases enormously.

What is that risk? It's the risk of MAINTENANCE. **If maintaining the TA is more expensive than manual testing, your TA has become a money eater instead of a money saver.**

Besides, E2E TA is usually linked to the UI, and that adds another instability factor, because the UI is usually more prone to changes than the back end.

Brain Positioning

In a majority of the cases, E2E TA automation in a start-up environment is a BAD idea. Management often doesn't understand what they are doing when they press the TA engineer to "automate it all." It IS possible to automate a lot, but tested software will surely go through many changes, so in the blink of an eye that smoothly working TA automation can turn into total junk or a time-consuming monster.

SL Let's look at the TA at Test Portal>Automation Scripts>Test Search. Here is what the `test_search.py` script does:

- First, it retrieves the HTML page with search results by submitting the following URL:
`http://www.sharelane.com/cgi-bin/search.py?keyword=moon`

- Next, it gets the expected keywords from the file:

`http://www.sharelane.com/test_search_ER.txt`

- Finally, it checks to see if the expected keywords are present within the HTML code of the page with the search results.

This is an extremely simple case, but even here we have a number of possible issues. Look at the URL:

`http://www.sharelane.com/cgi-bin/search.py?keyword=moon`

Brain Positioning

- **IF** the name of the script (`search.py`) changes, *the TA will not work.*
- **IF** the name of the variable "keyword" changes, *the TA will not work.*

- **IF** no books with "moon" in the title are present in the DB, *the TA will return the message "FAIL" even though it doesn't mean there is something wrong with search!*

- and most importantly, **IF** some of the keywords from the expected results (e.g., "Comments") are legitimately removed from the HTML code (e.g., because the feature containing those keywords was removed) *the TA will return the "FAIL" message even though it doesn't mean there is something wrong with search!*

Please note that we have just seen the potential problems in the case of an **extremely simple scenario**. What if the scenario is complex?

BTW, we've just learned about another side of TA – when the **TA produces an error, the TA engineer needs to spend time to determine whether that error is about:**

- **A bug**
- **A problem with the TA**
- **A change in the tested software**
- **A temporary issue with the tested software** – e.g., the Web server was down during the TA

run

Brain Positioning

In majority cases, an error produced by the TA does NOT mean a bug in the tested software.

If the TA fails by any reason *except* a bug in the tested software, all our efforts to fix the problem are called MAINTENANCE.

- If we are trying to determine what the heck is wrong with the TA, it's MAINTENANCE.
- If we ask the PM if there was a change in software, it's MAINTENANCE.
- If we even think about TA errors, it's MAINTENANCE.

Let's get back to the example with test_search.py. Now you are aware of at least some (!) risks for change even during that short, simple test flow. Does it make sense to automate it? After all, search IS a single component! I would personally give some thought as to whether it makes sense to automate even this piece. But the important point here is that **E2E test flows are usually made up of several components integrated with each other, so it's about the length of the road that your TA must walk through to get the actual result**. If the TA of even one component ("do search") looks like a questionable idea, think about a test flow with 2, 3, 4, 5, or more components! If we test using the UI, each component may have 1 or more Web forms to submit, so the length of the road may increase dramatically!

The only good approach I have discovered for E2E TA in start-ups is to create a short set of simple test cases for the basic flows, then automate that set using a similar approach (as in the case of Test Search), plug it into the test framework, and run that E2E TA as a smoke and acceptance test. The sad thing is that even this basic E2E TA will require your constant attention. How constant? Probably during every release...

BTW

As a rule, more bugs are found while writing automation scripts than while running them – i.e., during the several days it takes you to write an automation script, you'll probably find more bugs than your script will find during the coming months, or maybe even years.

BTW

Here is another note about MAINTENANCE I want to mention: MAINTENANCE is very demoralizing:

- A. For TA engineer
 - B. For those who depend upon the proper functioning of TA – e.g. the black/grey box testers who delegated running some scripts to TA.
- A. It feels like a real hassle when you constantly have to go back to what seemed to be finished. Think about how you would feel if you had to go back to high school and retake every examination each time a new set of exam questions is sent to your high school by the state.

Every TA engineer will inevitably encounter situations where he or she has automated a task that was not suitable for TA. Life is life, and TA is a very subtle subject, so mistakes do happen. My advice is this: **It's better to acknowledge your mistake ASAP than to waste your time (and thus your company's money) for the MAINTENANCE of poorly planned/designed TA.** A TA engineer who doesn't acknowledge his or her mistake and continues to do MAINTENANCE is like a person who has committed a crime and then has to constantly commit consequent crimes to cover the tracks of the original one.

A. Think how would you feel if someone who is more qualified than you and making much more money than you is, in fact, less productive than you. The failures of TA engineers are most evident to black box testers, because black/grey box testers are those who often get their test cases back with this remark: "Sorry, guys, our TA doesn't seem to handle your test cases, so start executing them manually like in the good old days." (The "good old days" were most likely when there was no "Let's automated it all!" nonsense in our company.)

BTW

Another aspect of maintenance is that parts of your TA can depend on code specifically created for your TA by programmers. Therefore, when tested code is changed, you might need to ask the programmers to fix that code. This way MAINTENANCE might take an extra toll by punishing the developers for their kindness.

So now you have a basic understanding about helpers, scripts for component automation, and scripts for end-to-end automation.

Now let me share some points on how to do TA right.

Let's look at the questions you must ask yourself BEFORE starting to write TA.

- a. How stable is the tested piece of software?**
- b. What are the frequency and length of manual execution of the candidate for automation?**
- c. What is the priority of the test suite/case to be automated?**
- d. How much longer will that task be around?**
- e. How hard will it be to write TA?**

a. How stable is the tested piece of software?

We've already seen that TA can be written only against stable piece of tested software. Otherwise maintenance will bite us in the butt every single release.

b. What are the frequency and length of manual execution of the candidate for automation?

Even if a piece of software is stable, it doesn't always make sense to write TA for it. For example,

IF

- We don't need to execute some task (e.g., test case #1) frequently (e.g., every release) AND
- The manual execution takes a short time (e.g., 10 minutes)

THEN

- It doesn't make sense to spend 1 week of expensive TA labor to write TA to automate that task.

c. What is the priority of the test suite/case to be automated?

TA is expensive. It doesn't make much sense to invest in the automation of a test suite/case that has a low priority (i.e., low meaning in terms of company business).

d. How much longer will that task be around?

If some feature is going to be removed from your software (or deprecated), it makes no sense to write TA for it.

e. How hard will it be to write TA?

Sometimes even easy trivial tasks (if they are performed manually) can turn into a TA engineer's nightmare. This is especially the case when UI is overloaded with scripting (e.g., JavaScript or Flash).

OK, let's proceed.

After the candidate for TA is carefully chosen, the next steps are to carefully design and carefully write TA. Again, entire books have been written about TA, and that is not the subject of this Course, but I want to give you some ideas to help you have good brain positioning on the subject.

There are two important factors about TA programming:

1. The architecture of TA
2. The technologies/tools used for TA

1. The architecture of TA

The scale of TA projects can vary substantially: in some cases, we can spend 30 minutes to create an automation helper (e.g., script `create_account.py`); in other cases, we can spend months writing test automation infrastructure. But in any case, TA programmers should always keep in mind the three "Es":

- A. Ease of maintenance
- B. Ease of expansion
- C. Ease of code reuse

A. Ease of maintenance refers to how much time should be spent updating the TA in the case of changes in the tested software. **Always remember that even the most stable component can be modified dramatically.**

Let's look at an example of a bad TA practice.

Example

Let's assume that we have 3 helper scripts:

- `automate_task_one.py` was written by Joe.
- `automate_task_two.py` was written by John.
- `automate_task_three.py` was written by Jason.

Each of these three tools has account creation as the first thing to do. So Joe, John, and Jason each wrote his own function to create an account. On one happy Friday ShareLane programmer Billy adds a new column to the DB table `users`: "age". The "age" column must be populated during account creation, because otherwise the account will be considered nonactive. As a consequence, ALL three helper scripts must be modified to comply with that change in the tested software. Does it look like Joe, John, and Jason have been thinking about maintenance of the first place? Probably not.

BTW

SL The solution to this particular problem is to create the function `create_account()` and place it into the module `qalib.py` (Test Portal>Automation Scripts>Test Automation Source Code>`qalib.py`). In order to create a new account, Joe, John, and Jason will simply need to import the function `create_account()` from `qalib.py` into their scripts. This way, in case of a change in the tested software (e.g., a new column in the DB, the TA should be changed in only one place: `qalib.py`).

You have to build your TA around the idea that tested software is going to change. While you are doing TA programming, consistently ask yourself *how easy will it be to modify your TA if the tested software is modified*. As we have already discussed, maintenance is the biggest concern with TA.

Example

Here is a good analogy about the relationship between tested software and TA:

In elementary school we used to make little sculptures of humans. The process was simple: first we would use metal wire to create a carcass, and then we would wrap something around it. So if I used clay as a wrapper, after the clay hardened I couldn't change the carcass position (e.g., make a sculpture of a standing person instead of a sitting person) without breaking the clay. But if I used Play-Doh that doesn't harden, that Play-Doh would easily and naturally change its shape as I changed the carcass position.

The moral here is that **your TA must be like Play-Doh: The TA must be designed to allow for easy and painless modification if the tested software is modified.**

BTW

In case of Test Discounts, it's really easy to change the TA if the product manager decides to change the logic between the number of books and discounts. We just have to spend 1 minute to modify the file with the expected results (test_discounts_ER.txt). Note that we don't need to modify the Python code (test_discounts.py) at all!

Another interesting fact here is that while designing test_discounts.py, **we expected that the most probable change** that could happen to function get_discount() would be a change in the logic between the number of books and the discount rates. So we designed test_discounts.py to make it easy to update it in case of a change in the tested software.

B. Here is another aspect of this TA business: How easily can you expand your TA?

Let's say that your TA does regression testing, and one day you identified several extra test cases that can be automated. Let's assume that the necessary functions like create_user() are already in place.

The question is this: how much time should you spend adding new automated test cases to your existing TA? In other words, how expandable is your TA?

- If your TA is poorly constructed, then adding new test cases can become a real hassle, because every time you might need to investigate... how your TA really works.

- On the other hand, if adding new automated test cases is turned into a routine, simple task, you just add new stuff on the fly without wasting your time constantly decrypting the TA code.

In the ideal situation, you just copy existing automated test cases, change some data in them, and welcome brand new members into your TA family.

Another aspect of expandability is whether or not it's easy to teach others to expand your automation. If your TA is written properly, it should not be a problem to create a tutorial/quick course to teach others about expanding your TA.

C. Code reuse is one of the cornerstones of good programming practices. Here are two aspects:

1. If you write your own code, you should always think about how reusable it will be for you and your colleagues.

2. In many cases, there is already code created by other people that you can reuse for your TA.

1. Code reuse means that existing code is used again and again to solve similar tasks.

For example, the `create_user()` function which we put into the module `qalib.py` can be reused every time the TA needs to create a new user account.

I can recommend the following things for better code reuse inside TA:

- It's better to create several simple functions than to pack code inside one big complex function.

- COMMENT YOUR CODE.**
- Create coding standards and adhere to them.**
- Don't try to get fancy; simplicity is the key.**
- Always think about those who might want to reuse your code.**

Code reusability is closely connected to both the maintainability and expandability of TA.

- Reusable code is usually well documented, well designed, and clean, so it's easier to maintain.
- Reusable code is a set of building blocks that can be used for future TA expansion.

2. For most people it's more fun to program than to search for code that has already been written and can be reused. But we should always remember that for everything we do, there is a chance that somebody has encountered the same (or a similar) task before and has already written solution for it. A very good example is the FREE test automation framework called DejaGnu. DejaGnu has been successfully used at PayPal QA for many years. The beauty of this framework is that you can write test cases in any programming language and just plug them in!

2. The technologies/tools used for TA

There is a huge array of technologies and tools that you can use for TA. I prefer to use the FREE and popular scripting languages (mostly Python and Ruby) under the Linux platform to do all my automation.

Let's compare two things:

- TA using free programming languages (**FL**) like Python, Ruby, Perl, and PHP

and

- TA using commercial tools (**CT**) like SilkTest and WinRunner

I'll list only the most important points here.

Reason number	Reason to use FL	Reason not to use CT
1	FL interpreters are free software.	CT can cost tens and hundreds of thousands of dollars.
2	When you join a new company the tools for creating TA (e.g., Python interpreter and Linux utilities) are already installed on your machine (or can be installed in a matter of minutes). The company doesn't need to buy you any special software. Thus, you can get started with your TA right away.	As a rule, automation engineers working with CT specialize in one particular CT. So, if your company doesn't have the particular CT preferred by the automation engineer, it must be purchased. Without that particular CT, the automation engineer is like a guitar player without a guitar.
3	<p>Working with FL, you expand your programming skills with popular and powerful programming languages.</p> <p>Those skills will surely be appreciated by all potential employers during your next job hunt. Why? Because:</p> <ul style="list-style-type: none"> - Everybody in technology is familiar with Python, Ruby, Perl, and PHP. - 99% of all Web companies use at least one of those languages to develop their software. 	<p>Working with CT, you expand your skills in that particular CT because each CT:</p> <ul style="list-style-type: none"> - Usually has its own programming language <p>and</p> <ul style="list-style-type: none"> - Contains lots of nuances SPECIFIC to that particular CT <p>Your skills will be appreciated only by those employers that have already purchased (or plan to purchase) that particular CT.</p> <p>BTW Ask any software developer what "4Test" means (4Test is SilkTest's own language – no other software uses it). He or she will probably reply: "What the heck are you talking about, my friend?"</p>
4	If you want to transfer your TA knowledge, you just create documentation on the Wiki. There's no need to explain how to write code in the FL that you've used (e.g., Python).	If you want to transfer your TA knowledge, you have to specifically teach another person the programming language used by that concrete CT AND all the nuances relating to that concrete CT. That's just the first step. On top of this, you'll also have to create documentation on Wiki, just like with FL.
5	When you leave the company, your TA can be expanded and	When a CT engineer leaves the company, there's not much hiring

	maintained by anyone who knows the FL that you've used for TA (e.g., Python).	flexibility – a particular TA has already been purchased. The company will need to hire a TA engineer who specializes in that particular CT.
6	On the Web, there are thousands of free, well documented programs written in a FL you can use for TA.	On the Web, you will not find much code for CT.
7	The Web has an enormous number of forums dedicated to FL, because millions of people use them. This means that the majority of the issues/questions that you'll encounter using FL have already been discussed somewhere. In 99% of the cases, you can find your answers within seconds using Google search. You can also get help from the developers inside your company because they use the same (or similar) programming languages.	CT are expensive, thus not many people are knowledgeable about them (compared to FL). If you get stuck on something during CT usage, good luck finding an answer on the Web. Most likely, your developers will not be able to help you either; knowledge about CT usage is very specialized. In addition, you might need to pay the producer of CT for support. But even if CT support is free, you'll probably have to wait until some support dude is available to answer your questions. BTW Note that once that CT support dude becomes available, there's no guarantee that he is qualified to answer your question. I remember a situation where my manager had to go all the way to the CTO of one famous CT producer to get adequate support for \$50K(!) CT used for performance testing.
8	You company doesn't need to pay for extra copies of Python interpreter if you want others to expand your TA.	Extra CT users mean extra CT licenses; extra CT licenses mean extra money; extra money means an increase in cost – an increase in thousands or even tens of thousands of dollars.

I hope that my reasons make sense to you (and will make sense to your employers!)

To be fair, I must admit that there ARE success stories about creating test automation with CT; I just haven't heard very many of them. And when I hear admiration about some concrete CT, usually it's not about how reliable and maintainable the CT scripts are in the long run, but about cool tricks that can be written under the CT environment. But here we are not talking about TA as a way to impress our little sisters with wonder tools that magically fill up Web forms inside Web browsers. We are talking about TA as a way to save money for our companies, and I feel that my anti-CT reasons make sense in most cases.

BTW

There is a huge temptation among beginner testers to specialize in SilkTest or WinRunner or some other CT. My recommendation here is this: If the company where you work has one of these tools, use the opportunity to learn that CT. Otherwise, don't waste your time and money. It's better to install Python interpreter and Cygwin on your Windows machine and start learning Python scripting and Linux commands.

Job market for the test engineers with Python (or other FL) + Linux skills is MUCH bigger than for those who specialize in CT.

In the future, I'm planning to write a QA course on TA with Python. But for now, I can give you one simple – but very valuable – idea on how to do UI automation with Python. BTW, we used that idea when we created `test_search.py`.

In a nutshell, the idea is:

1. Use the Linux command 'wget' to submit Web forms and retrieve HTML pages.
2. Use Python as a wrapper around wget to drive wget and parse the HTML code of the retrieved pages.

Look inside the code for `test_search.py` and you'll get it. Tool wget can be installed under the Cygwin environment on your Windows machine.

When Regression Testing Stops

Basically, there are two main factors here:

- The deadline set for RT completion
- The test cases to execute for RT

The challenge is to execute all the test cases that we decided to execute for RT before or at the deadline set for RT completion. In many cases, this involves overtime work.

Brain Positioning

Overtime work is a part of our profession, so it shouldn't come as surprise to you if you have to work more than 8 hours a day to finish whatever you do – e.g., the execution of a particular test suite. As far as I know, the most successful testers are those who don't count hours at the office, but go above and beyond to do an excellent job. In some cases, an excellent job requires long hours.

Let's proceed.

I recommend setting a time frame of several days between the deadline set for RT completion of a major release and the Go/No-Go meeting. That time can be spent to fix/verify bugs that haven't been fixed/verified during RT and to do an acceptance testing (also called a "certification testing"). Once the acceptance test has passed, we get together for the Go/No-Go meeting to decide whether we want to release or not. If we don't want to release, then we must resolve the issues that prevent us from releasing,

do another acceptance test, and get together for the Go/No-Go meeting again – and so on until we do our release.

BTW

In many companies, the Go/No-Go meeting for major releases is held on a particular predefined date, so there is no dependency on whether RT and acceptance testing are finished or not. In that case, at the Go/No-Go meeting we just discuss the current testing (and bug fix) status and decide on whether we are going to release or not.

Lecture Recap

1. We need regression testing because:

- a. It's often extremely difficult for a programmer to know how change in one part of the software will echo in other parts of software.

AND **what's even worse...**

- b. Programmers sometimes change software **without even trying** to figure out if their changes will break something.

2. Testers usually know (or can make an educated guess) which features will be directly affected by a change.

3. The **1st group** of test suites for regression must contain the test suites checking legacy (i.e., existing) features that are **directly affected** by a change in the software.

4. The **2nd group** of test suites for regression must contain the test suites checking legacy features that may somehow **depend** on the changed features.

5. The **3rd group** of test suites for regression testing contains the test suites checking legacy features that don't fall into the first or second group.

6. The fact that a test suite falls into the 1st, 2nd, or 3rd group doesn't mean that we are going to execute it. For example, we might be short of time.

7. Below are the ways to settle the contradiction between our limited resources and the ever-growing number of test suites:

- a. Prioritization of test suites and test cases
- b. Test suite optimization
- c. New hires or outsourcing
- d. Test automation

8. Prioritization of test suites and test suites/cases is the most inexpensive and (in most cases) the most effective way to resolve this issue.

9. TA for regression testing comes in two forms:

- Helpers
- Automation scripts

10. Helpers are loved by everybody, because they are a real help in the manual execution of everyday routine tasks. Sometimes testing is not possible without helpers (Credit Card Generator).

11. Automation scripts come in the form of:

- Tools for component automation
- Scripts for end-to-end automation

12. The number one enemy of TA is MAINTENANCE.

13. TA maintenance has two roots:

- Start-up software is always changing, thus the TA must be changed too.
- TA can be poorly designed, so even a small change in the tested software can affect the maintenance of that TA.

14. Helpers are less prone to maintenance than automation scripts, because they don't test software; they automate tasks around testing.

15. Component automation must use **short, isolated** test flows in order to be less prone to maintenance.

16. In most cases, end-to-end automation in start-ups is a BAD idea.

17. In majority of cases, if the TA produces an error, it doesn't mean that the software has a bug.

18. If TA produces an error and that error is not related to a bug in the software, then time must be spent on maintenance.

19. Good TA saves money. Bad TA is a hungry monster that can eat up hundreds of thousands of dollars with nothing or little in return.

20. Questions to ask before automating something:

- a. How stable is the tested piece of software?
- b. What are the frequency and length of manual execution of this candidate for automation?
- c. What is the priority of the test suite/case to be automated?
- d. How much longer will that task be around?
- e. How hard will it be to write TA?

21. There are 2 things that affect the end of RT:

- The deadline set for RT completion
- The test cases that we have decided to execute for RT

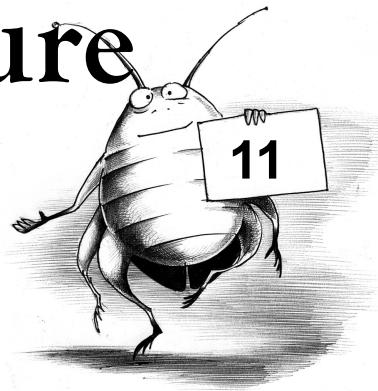
22. The challenge is to execute all test cases that we have decided to execute for RT before or at the deadline set for RT completion.

23. The decision about whether to release or not is made at the Go/No Go meeting.

Questions & Exercises

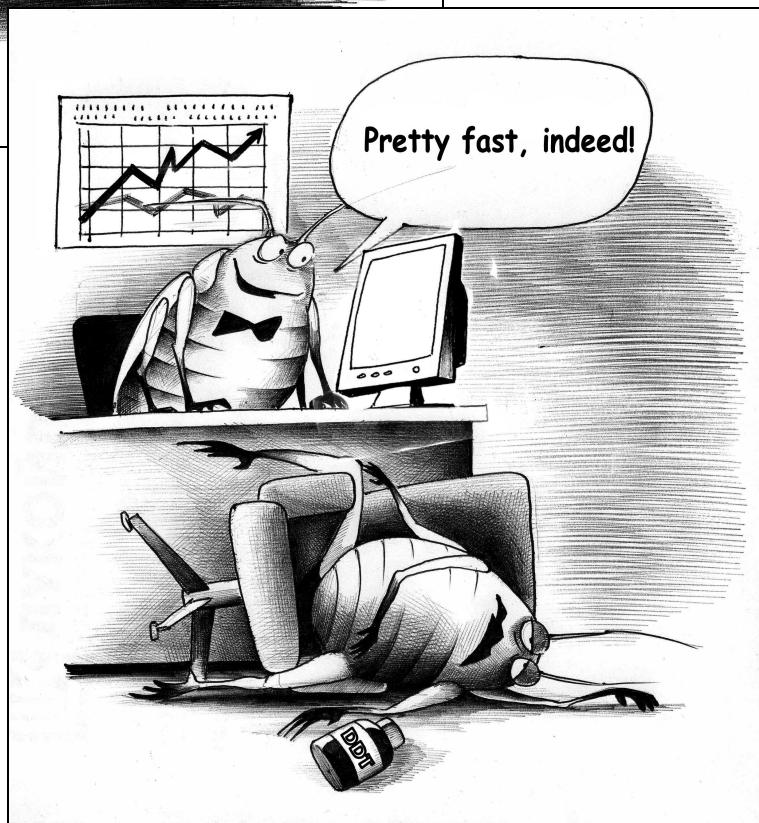
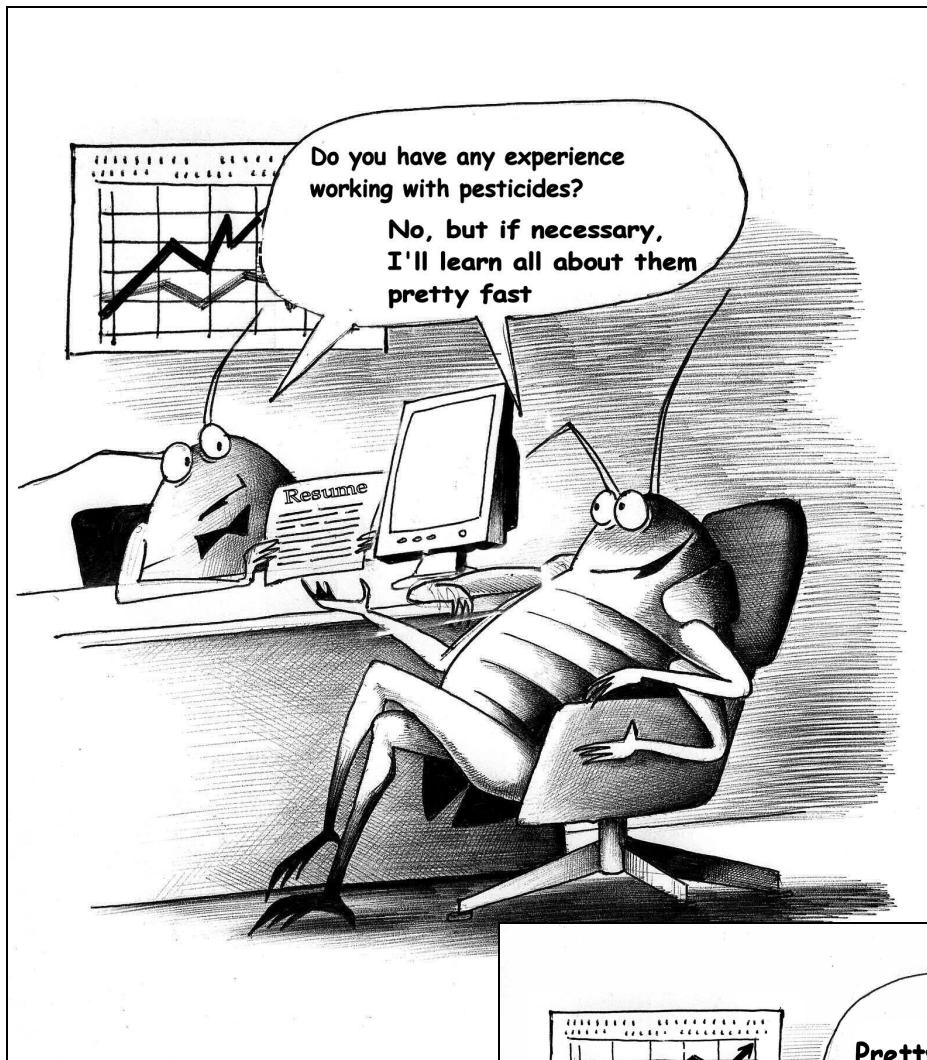
1. Why do we need regression testing?
2. List and describe 3 groups of test suites for regression testing.
3. If we have one test suite from the 1st group and one test suite from the 2nd group, and there is time to execute only one of them, which test suite would you pick up?
4. Describe some ways to settle the issue of limited resources and the ever-growing number of test suites.
5. What is a helper?
6. What does the concept of "the length of the road" have to do with TA?
7. What is the difference between helpers and automation scripts?
8. What kind of automation script is less prone to maintenance? Why?
9. Why is E2E automation usually a bad idea for Web start-ups?
10. What are the causes of TA maintenance?
11. What questions should you ask before automating something?
12. What are the pros for using FL for TA in Web start-ups?
13. What are the cons for using CT for TA in Web start-ups?
14. When does RT stop?
15. What decisions can be made during the Go/No-Go meeting?

Lecture



How to Find Your First Job in Testing

Lecture 11. How To Find Your First Job In Testing.....	301
Why You Have A REAL Chance To Find A Job In Testing.....	303
Mental Tuning.....	304
Job Hunting Activities.....	306
Lecture Recap.....	336
Questions & Exercises.....	338



"The brick walls are there for a reason.
The brick walls are not there to keep us out.
The brick walls are there to give us a chance to show how badly we want something.
Because the brick walls are there to stop the people who don't want it badly enough."
- Randy Pausch

If you want to do something,
you will find 1000 ways to do it.
If you don't want to do something,
you will find 1000 excuses not to do it.
- Nikita Toulinov

Why You Have A REAL Chance To Find A Job In Testing

Before I got my first job in testing, I was given two types of advice from those who already had a job:

- It's easy to find a first job in testing.
- It's almost impossible; don't even try.

I rejected the second piece of advice on the fly, because I don't take advice from people who use word "impossible."

I listened to the first type of advice, which also proved to be wrong. **It's not easy to find a first job in testing.**

It's not like thousands of software companies are just waiting for your signal: "Okay, I'm available. Send me your job offers. Don't offer me less than \$120,000 a year."

But the reality is that **there are thousands of companies where you can be valuable as an entry-level tester, and there is a REAL chance for you to find a job in one of those companies. THERE WILL ALWAYS BE A NEED FOR AN INEXPENSIVE YET SMART AND MOTIVATED WORKFORCE IN SOFTWARE TESTING.** The challenge for employers is that it's hard to find this type of employee. My target for this chapter is to show you **how to make yourself visible and convince an employer that you are valuable.**

Just keep the following in mind:

- Finding your first job is not an easy process.
 - It can take weeks or sometimes months before you get your first interview.
 - Most likely you'll have lots of disappointments, rejections, and hurt feelings.
- BUT
- If you'll be smart, persistent, and patient, you WILL find a job.

Why is it hard to find a first job? The answer is simple – companies need candidates who can be productive ASAP. It's also obvious that those working as testers now once didn't work as testers; at

some point, they did something to get those positions. So there is light at the end of the tunnel, and I'm here to tell you that **getting a job in software testing with zero experience is absolutely a realistic thing.**

Now, let's get this straight: realistic doesn't mean "easy" or "simple." It means that **if you really want it and are ready to work hard and smart, you'll find that job.**

Mental Tuning

Do you know what REALLY prevents a person from getting an entry-level job in testing?

- It's NOT a lack of experience.
- It's NOT a lack of connections.
- It's NOT a lack of education.
- It's NOT a lack of communication skills.

IT'S A WRONG ATTITUDE. So – what is the definition of a wrong attitude? There are many flavors, but two of the most common are these:

- **Negativity and skepticism:** e.g., *"Oh, yeah? Do you think you're a smart ass? Every available entry-level position is taken by somebody's friend or relative,"* or *"You cannot just come to a foreign country and make more money than the majority of locals,"* or *"Come on, man. You cannot jump over your head."*

- **A strong desire to make good money combined with a complete lack of any desire to work hard:** e.g., *"Wow, you testers are making pretty good money! Oh, are you saying I'll have to sacrifice watching the Sports Channel and partying with friends? Nah, that doesn't sound right."*

As you've seen from previous chapters, **software testing is not rocket science.** There is a lot of stuff to learn when you encounter testing for the first time, but in the end this stuff is totally comprehensible. I've seen folks with very primitive computer skills who were able to pick up testing techniques and get a job. But technicalities are the simplest part of a tester's education. **The hardest part is changing a student's attitude if that attitude is wrong.** And here I often find myself powerless because **strong wrong attitudes usually have deep roots in an individual's personality**, and that I cannot change.

On the other hand, I've known many guys and gals with dead-end jobs who were working hard on their testing education and job hunting. Guess what? Those who really wanted to be employed ARE employed. Was it easy for them? NO. But they had right attitude (I'll define "right" in a minute) that helped them to achieve their goal of changing their lives and getting a job in the software industry.

This is very important: **For your first job, your attitude is of primary importance.** After all, what else can you offer if you have zero previous experience? Why would an employer hire you? My answer is that smart employers know perfectly well that a **right attitude for an entry-level tester converts into actual work results in a matter of days or weeks because that person will do his or her best to justify that employer's trust!**

What is the RIGHT attitude? **The gist of having a right attitude is simple: GIVING.** Giving your time, your efforts, and your talents – to your search for a new job and to the company that hires you.

Yes, you need a software testing job to **receive** (money, stability, etc.), but **you'll receive a lot only if you give a lot in the first place.**

Here are the three parts of the mantra for anyone who wants to get his or her first job in testing:

- "I'm willing to work unlimited hours."**
- "I'm willing to work on weekends and holidays."**
- "I'm willing to work for any amount of money."**

I'm still using the first two lines in my own interviews, because I believe that this is required of every test engineer, and it generally makes a candidate very attractive independent of the level of his or her professional expertise.

The third line is needed for the entry-level job candidate to show his or her deep, sincere desire to get the job. This also lets a potential employer know that you want to give rather than receive.

Now, forget about software testing for a moment. Imagine this situation:

You need a housekeeper to clean your apartment, cook for you, send your clothes to the dry cleaners, and walk your spaniel. What would you say if someone in good faith offered you all these services for ANY fee that you were willing to pay? Well, anyone in his or her sane mind would ask the following question in response to this kind of offer: "What's the catch?"

*The catch is that you will give that employee a chance to apply **in practice** whatever he or she already knows **in theory**.*

- Does this sound like a good deal for you? Of course: not only will you get a good deal for housekeeping services, but you'll also do a good deed by giving someone an opportunity to gain experience!
- Does it sound like a good deal for the employee? Of course: he or she will be able to gain experience and make some money at the same time.

Now imagine:

The individual has worked for you for six months and has become really skilled in all areas of your housekeeping needs. Would you let him or her go just to get another worker with the same experience? Of course not! But if there is a special situation (like you move from San Francisco to Madrid) and you have to part with your housekeeper, then surely you'll try to find that person a job at your friend's house, or at least you would give him or her the best possible recommendation!

Why don't we translate this situation to software testing jobs, replacing a housekeeper with a beginner software tester and the homeowner with the QA manager? Your great advantage is that not many people approach QA managers with offers like this:

- "I'm willing to work unlimited hours."**
- "I'm willing to work on weekends and holidays."**

"I'm willing to work for any amount of money."

Why? Because it's an out-of-the-box approach, and inexperienced people are scared to do things that are out of the ordinary. I personally took this unconventional approach, and I know others who did too and got the same amazing results: **hiring managers were shocked and...helpful.** It wasn't all smooth and easy, but in the end, opportunities were given to us, and experience was gained. Now my colleagues and I have a different problem: how to respond to all the emails we get every day from recruiters.

To finish this section on mental tuning, let me tell you this: **The wrong attitude of an otherwise great candidates is the main reason why the attractive opportunity called "entry-level job in software testing" will always be available to those who have the RIGHT attitude.**

Now, let's get down to the step-by-step instructions for job hunting.

Job Hunting Activities

ACTIVITY 0: Tune up your attitude. You are not the first or the last person looking for a first job. However, if you have the right attitude, your chances will improve tremendously.

Main action of ACTIVITY 0: **Set your mind in the right direction.**

ACTIVITY 1: Let people in your network know that you are looking for an entry-level position in testing and are ready to work unlimited hours, even on weekends and holidays, with minimal pay.

During one career-related seminar, the lecturer told us something very interesting: "Networking is not simply a **desirable** thing anymore. **Nowadays, networking is a necessity.**" What is networking? It's a set of activities designed to meet new people, make contacts, and maintain those contacts.

BTW

The most common networks are:

- our friends and relatives
- our classmates from high school and college
- our former and present co-workers

The lecturer's point was that **we have to consistently work on widening our network, and as we do, our opportunities for success in life increase.**

Who would a hiring manager contact first if he or she needs to hire a tester? That's right – people from his or her network! So, if you are looking for a job, and a member of your network is looking for a candidate, this is a perfect situation for both of you!

But what if the people inside your network don't work for software companies? That's a very likely situation, but think about this. Let's say you have ten friends, and each of these friends has ten friends. This means that if you ask ten of your friends to ask their friends, your message can possibly reach 100 people! So even if ten of your friends don't work for software companies, they may know somebody

who does. In reality, we all have many more than ten contacts, so your chances of finding a job via your network are much better than you think.

BTW

It's proven that in the majority of cases, **there are no more than six degrees of separation between any two persons in the world**. *One degree is the virtual distance between you and someone you know personally; e.g., there is one degree between you and your friend John, and there are two degrees between you and John's acquaintance Helen.* Think about it: there are no more than six degrees of separation between you and **anybody** in this world. So all the hiring managers in the world are much closer to you if you think from this perspective!

The best tool to formally manage and expand your professional network is LinkedIn.com. This project became so important in the business world that it's been said, "If you are not on LinkedIn, you don't exist." Whatever professional stage you are at now, create an account there and start collecting your contacts.

1. Find and connect to people you know now.
2. Send invitations to join your network to people you meet.

Here are the benefits of using LinkedIn:

1. You'll have a presence at a global networking place.
2. People you know can recommend you to people you don't know.
3. There is a strong chance that a hiring manager will look at your LinkedIn profile before inviting you for an interview.
4. People who belong to your LinkedIn network can write recommendations for you that everyone can see in your profile. The more recommendations you have, the better your chances to get invited for an interview.
5. There is a feature called "People in your network who are hiring now" where you can see open positions.
6. In February 2008, LinkedIn introduced many new job hunters' tools.

There are three types of *natural* networks:

- your professional network, i.e., your colleagues
- your school network, i.e., your classmates from school and college
- your social network, i.e., your friends and relatives

You should "link in" to the people from all of these networks, because all of them can be useful for your career.

There are other networking sites like Facebook that help people to meet people and make useful contacts.

But no network will help a person in the long run if he or she has a poor attitude, EVEN if he or she is very good professionally! LinkedIn is just a promotional tool, and no promotion can help in the long run if what's being promoted doesn't meet the expectations of consumers – i.e., companies or people you'll

work for. That's why it's imperative to understand that your value comes from WHO YOU ARE and WHAT YOU CAN (OR ARE WILLING) TO DO in the first place. Your network is just a helpful instrument to give you more chances to demonstrate your value. That's why I keep emphasizing the importance of attitude!

When you want to send a message to your network about your interest in obtaining an entry-level position, it's very important to contact the right persons. Here is what you should do: browse through

- your LinkedIn contacts *and*
- the contacts of your contacts,

looking for those who work for software companies or recruiting agencies.

- If somebody in your network is in the software or recruiting business, email them, have lunch with them (treat must be on you!), call them, etc., to let them know about your interest in being a software tester. **Always have your resume ready, and don't be shy about emailing it** (we'll talk about resumes in a minute).

- If a necessary contact is not in **your** network, ask a person from your network to recommend you. This all can be done using LinkedIn features.

And of course, you should talk to your friends and the people you meet about your intentions of entering the software industry: very often destiny creates great surprises for those who are looking – e.g., at the party you might meet somebody who works at a software company that currently is hiring testers.

Main actions of ACTIVITY 1:

1. Start to organize and expand your network.
2. Begin sending a message about your intentions to become a software tester.

ACTIVITY 2: Create a resume. This ACTIVITY should take place concurrently with ACTIVITY 1.

A resume is an advertisement. The recipients of that advertisement are employers and recruiters. Here is the difference between TV advertisements and a resume:

- On TV, viewers can be **endlessly** brainwashed until zombification is accomplished.
- In the case of a resume, there is no brainwashing – your resume has only **one chance** to make a good impression and inspire its recipient to call or email you.

A resume is the presentation of your knowledge and your virtues. The word "presentation" has a primary meaning here. The main difference between successful and unsuccessful resumes is the effectiveness of the presentation.

Your presentation is effective if the recipient

- 1. Completely understands the message expressed in your presentation**
- 2. Acts according to how you expect him or her to act**

During a job hunt, the recipients are employers and recruiters.

1. Your message is your interest in employment. You want to communicate this message properly.

On the one hand, in your resume you should clearly express:

- **what you want:** an entry-level tester position;
- **what you can offer:** your attitude, dedication, and acceptance of low pay;

On the other hand:

- **Your resume should be very well written.** I doubt anyone would be interested in a candidate whose resume is full of grammatical mistakes and misspellings.
- **Your resume should use an energetic and confident tone** – e.g., if you write “Achieved 10% reduction in expenses by introducing innovative technologies,” this sounds much better than “I was good for my company by changing the way they did things.”
- **Your resume should be nicely formatted** so the eye of the reader can immediately spot the necessary info. Appearance is the first thing that catches someone’s attention.

2. Your resume should motivate an employer or recruiter to call or email you.

Here are some practical things you can do to implement all these ideas about creating an effective presentation.

STEP 1, create a list of:

- your achievements**
- your testing-related experiences**

Let’s talk about **achievements**. A good tester is not just a person who knows how to test code and file bugs.

- A good tester is an achiever.
- A good tester is a person who produces results.
- A good tester has an I GIVE attitude.

You want to show recipients of your resume that YOU ARE ABLE TO ACHIEVE STRONG RESULTS. In fact, each of us has something to remember in this department. *For example, one of my pals worked at an electronics store before becoming a software tester. He was one of the top salesmen, and he naturally put that info on his first resume. Guess what? It worked!* Even if you just graduated from college, surely you can share some cool projects and situations where you took the initiative and did something extraordinary!

Of course, you won’t be able to remember all this at once. Put a piece of paper in your wallet and write down your achievements as you think of them: on the train, during lunch, walking in the park, etc. Create a comprehensive dirty list by writing down everything that you remember. Don’t worry about filtering it now.

Next, you want to **rethink your experiences from a testing-related point of view** and present your findings in professional way.

Example

Wendy works at a print shop. One day, a new binding machine arrived from Germany. Wendy single-handedly set up this machine and taught others how to use it. Should she write on her first testing resume: "Single-handedly set up binding machine and taught others how to use it"? Nope. This description sucks. Let's restate it in a more effective way:

"Created and executed test procedures to detect setup problems with new equipment."

"Organized and conducted training seminar on new binding technologies."

Creating and executing test procedures was just a part of the setup activities for the new equipment. But Wendy's purpose is to showcase her testing abilities and demonstrate the correct usage of professional language, rather than boring her recipients with technical stuff related to the print shop.

Rethinking your experience from a testing-related point of view can be challenging at first, but once you start writing down things down (dirty list), your memory will recall lots of useful items.

BTW

Wherever you work, make sure to keep records of important things that you do for your company. I do it using a text file that I've named done_stuff.txt. There are two important things here:

1. Don't forget to update the file.
2. Use good business language when you do your updates.

Let's say you spend time explaining to your pal Alberto how to create and verify credit card transactions using Helpers from ShareLane Test Portal. Do you know how to really describe your actions? Not "Helped Alberto to dig into stuff," but "Performed personnel training on credit card testing." You can easily use this entry as one reason for your promotion!

Many of your colleagues have this type of file and accurately keep it up-to-date. Here is an example of done_stuff.txt:

Apr. 07 – Apr. 11

Performed personnel training on credit card testing

Wrote helper tool view_time.py (UNIX Time Utility)

Completed regression testing for "Checkout with credit cards" (TS7122)

Apr. 14 - Apr. 18

Interviewed candidate for QA position

Organized meeting to discuss improvements in Bug Tracking Process

Updated test cases for "Tests for Rewards Program" (TS8341)

The main reason to keep this file is that people tend to forget. You can forget about some of your

accomplishments (**especially if you have many of them**) and other people (including your manager) can forget about your accomplishments as well.

Another reason is that this saves you time in case you have to write a status report or fill out a quarterly self-evaluation form.

Make a habit of updating this file, and you'll never regret it.

Once you have a dirty list, it's time to sit down and modify it into a white list. Use this format:

<Name and location of previous employer, e.g., *Some Company, GMBH. Munich, Germany.* (Put name of your college if you just graduated and have zero professional experience in any area)>

1. <First achievement or testing related experience, e.g., *Created and executed test procedures to detect setup problems with new equipment*>
2. <Second achievement or testing related experience, e.g., *Organized and conducted training seminar on new binding technologies*>

Once you have the first version of your white list:

1. **Modify the sentences of your white list so they start with action verbs and use powerful adjectives.**

Verbs: Accomplished, Achieved, Arranged, Built, Clarified, Created, Defined, Designed, Developed, Discovered, Documented, Established, Implemented, Improved, Increased, Introduced, Invented, Maximized, Optimized, Organized, Pioneered, Produced, Promoted, Provided, Reduced, Saved, Standardized, _____ (write some more of your own)

Adjectives: Active, Effective, Important, Innovative, Outstanding, Positive, Powerful, Skilled, Strong, Substantial, Successful, Unique, Valuable,

_____ (write some more of your own)

2. **Try to remember concrete figures and numbers; in other words, measurable indicators of your achievements.** For example, "Successfully introduced new processes that increased production output by 20%" sounds more convincing than an immeasurable "Successfully introduced new processes."

STEP 2: Put your white list into your resume template. Download the template for a first resume from qatutor.com and fill in the section **Professional Experience** with entries from your white list.

STEP 3: Get some experience as beta tester and populate the section Beta Testing Experience. As you know, employers want testers with experience. So get some experience! It's much easier than you think: Just Google the phrase "beta testers wanted" and you'll find quite a few projects where you can

contribute. Also, check out Google Labs; they always have something brewing there, e.g., as of February 2008, you can contribute by testing *Experimental Search* and *Image Labeler*.

Select several projects you like; if needed, become an official beta tester and start testing. **Don't be discouraged if it takes some time for you to find good projects to beta test; just keep looking and communicating, and you'll eventually find what you need.**

Try to expose yourself to as many aspects of beta testing as you can. For instance, it would be great to have direct access to a bug tracking system and be able to communicate with its developers, but this kind of situation is rare. Usually you just send your bug reports to an email address or use a special Web form to report bugs/concerns/issues/etc.

Dedicate some time every day to beta testing. This is good for your testing experience *and* for your resume! **Make sure you write down your experiences**, e.g., bugs that you find, recommendations you come up with, etc., to create a foundation for material you can put into your resume. Once you have enough stuff to write home about, create a white list of your beta testing experiences and enter it into the **Beta Testing Experience** section of your resume.

I want to stress two things about your beta testing:

- **Don't be shy about reporting problems, and don't be afraid that the problems you discover will be made fun of.** You know that you don't have much experience in software testing, and those who invited you to do beta testing are also fully aware of the fact that they are dealing with nonprofessional testers. Go ahead and report any problems you see and are able to reproduce. If you take this beta testing seriously, your professional skills in bug finding and reporting will grow tremendously, because you'll be dealing with real projects and doing a real testing job.

- **Show your attitude.** You probably won't get paid for beta testing, but if you're able to dedicate some undivided time (for example, one hour every day), then you'll do well for yourself and the company. It's likely that the company will never even send you a "thank you" email, but YOU'LL KNOW that **you took the job and did it properly**.

STEP 4: populate the section labeled Software Experience. We all hang out on YouTube, a social networking site, or whatever sites we find useful. Guess what? Mentioning this can be really good for your first resume, because they demonstrate that you are an active user of different Web projects and thus have **an experience dealing with various Web applications**. Also make sure to mention your experience in dealing with operating systems and specialized software, like Adobe Photoshop.

Example

Software Experience:

Advanced computer user with PC, **UNIX, and Mac experience**

Active user of **social networking Web sites**: MySpace, YouTube, FaceBook

Active user of the **mobile Web**

Advanced user of **electronic payments**: PayPal

eBay merchant since 2002, with **142 positive feedbacks**

STEP 5: populate all other sections of the resume – Name, Address, Personal Summary, and Education. Just list all needed information in a straightforward way. If, in addition to your college education, you have some relevant courses ("Introduction to UNIX"), make sure to mention these as well.

BTW

Here is quick note about the phone number you list on your resume. Please consider the following things:

1. Once you send your resume out to the world, it will take on a life of its own.

- You'll have little or no control over its content.
- It can easily become available to anyone.

Example 1: After you send your resume to John, he can send it to Mary, Sandra, and Steve, who might send it to other people, and so on.

Example 2: After you post your resume on some recruitment Web site, the Google crawler can dig it out and Google would save it as cached search result. So your resume can be found via Google search even after you remove your resume from that recruitment Web site.

2. If you decide to put your phone number on your resume, give your cell phone number rather than your home phone number – for two reasons.

Reason 1: You don't want a call from a recruiter or a potential employer to be picked up by anyone but you, e.g. by your younger brother who might be enthusiastically chewing a crunchy green apple at the time and saying, "Jason-boy" is not home. Who the heck are you?"

Reason 2: In many cases, it's much simpler to change your cell phone number than your home phone number. My friend is still getting calls from recruiters during family gatherings at the dinner table several years after he retired.

I just want to warn you about the possible exposure of your privacy. I personally don't ever put my phone number on my resume, but as far as I know, most people don't mind doing this.

STEP 6: polish the language of your resume.

In many cases, English is your second language. You might think that your writing in a foreign language is beautiful, but you don't want someone to look at your writing and say, "What is that?"

Example

I've been using English for many years, but when I first submitted one of my articles to my dear friend, writer John Fogg, he called me back and said that my material should be

- first translated from Roman's English to typical English, and then
- from typical English to book English.

So I had to look not only for a professional editor but one who had experience understanding written English used by foreigners.

Of course, a resume is not a book, but you get the point. What I recommend for resumes in English is this: post an ad on Craigslist.com with the title "20 bucks for quick editing job," and explain your situation in body of the message, e.g., "I need a native English speaker to go through my resume. Will pay 20 bucks." You'll get many responses within several hours. Make sure that person has a Web site or some other means to verify that he or she really can do the job.

If you *are* a native English speaker, it's still a good idea to give that resume to somebody for feedback. And in any case, **ALWAYS USE SPELL CHECKER!**

You'll need to have your resume in three formats:

1. Microsoft Word document
2. HTML file
3. Text file

First, create your resume using MS Word, and then save it as a .doc file, then as an .html file, and then as a .txt file. Easy.

Main actions of ACTIVITY 2:

1. Remember your achievements and testing experiences.
2. Do beta testing.
3. Fill out the resume template.
4. If applicable, get an edited version of your resume from a native English speaker.

ACTIVITY 3: Find recruiting agencies that look for professionals for software companies in your target market.

Your target market is a certain geographical area where you want to look for work, e.g., Silicon Valley or Bangalore.

Here are four recruitment sites that you want to be familiar with if you look for employment in the U.S.:

Craigslist.com
Dice.com
HotJobs.com
Monster.com

Let's call them The Big Four. If you are not looking for employment in the U.S., find recruiting Web sites in your country. This is not a hard task; just use Google search. *From now on, I'll assume that you are looking for a job in the United States, but you can use this same approach for any country.*

Let's say you go to HotJobs.com:

1. Using Search, enter the keyword "tester" (or "QA") and specify your target market, "CA" (California).
2. Browse through the list of search results, looking for recruiting agencies.
3. Put the agency name, Web site, and the city and state where the recruiter is physically located into a special file. (*You can get city/state info in the "Contact Us" section of the recruiter's Web site.*)

You can use this format:

Name	Website	City/State	Did I contact them? (Y/N)	Comments
Some Recruiting Agency, Inc.	www.<address>.com	San Francisco, CA		

Do this for all of The Big Four sites.

Look at your table. You have just created a list of your potential **business partners**.

Brain Positioning

Recruiters are basically matchmakers who try to connect software companies with potential employees (or consultants). The great thing about recruiters is that they are INTERESTED in your employment (or your contract) because they will get a nice commission out of it! Software companies often pay recruiters tens of thousands of dollars per candidate for a job, so it's understandable that recruiters are more than willing to help you in your search: IF YOU WIN, THEY WIN TOO!

Next, you'll want to establish a contact with those recruiters.

Let's look at the column "City/State." The best-case scenario is if the city/state of the recruiter is close to your physical location. Why? Because we need a list of recruiters in order to communicate with them, and the best way to do this is to meet with those recruiters in person.

BTW, depending on the context, "recruiter" can refer to either a recruiting agency or a person.

Recruiters are more than willing to meet you! So, call and set up a meeting with a person who is dealing with software companies. Tell him or her about your situation; show him or her your resume. Ask for advice and guidance. SHOW YOUR ATTITUDE! And remember that your success is their success; they are not helping you for free.

In other cases, a recruiter will not be within driving/train/subway distance. In this case, you can communicate with your recruiter via phone, email, or instant messaging.

Some very important things to find out from recruiters are: market condition, nuances, and trends.

For example, you might find out that there are many new start-ups that are developing software for mobile phones. What does this tell you? To go ahead and read about wireless technologies and about the nuances of testing of mobile software!

Don't be shy to ask recruiters what they think about your resume. They can give you some pretty good guidelines.

Make sure to add the persons who you meet personally or via email or phone to your LinkedIn contacts.

Once you establish connections with some recruiters, you have representatives who will be promoting you to software companies. Not bad at all, considering that you pay nothing for it!

In case you establish "enough" contacts with recruiters, make sure to email your resume to recruiters on the rest of the list (and/or create a profile on their Web sites).

Keep checking for new job postings, and update your list of recruiters every day. Looking for a job is also a JOB!

The least a recruiter can do for you is to give you a piece of advice.

The most a recruiter can do for you is to set up an interview with a potential employer at an actual software company.

Main actions of ACTIVITY 3:

1. Create a list of recruiting agencies.
2. Establish contacts.
3. Get advice about how to find a job.
4. Get feedback about your resume, and if needed, change it.

ACTIVITY 4: Launch a campaign dedicated to self-promotion.

Although there can be a great deal of help from the recruiters that you've met, don't just sit and wait for a call from them. You should start an AGGRESSIVE campaign to promote yourself.

Approach 1: Create a profile and/or post your resume on The Big Four*.

Approach 2: Start sending your resume to as many employers and recruiters as possible.

**Of course, you can use more than the four previously mentioned recruiting Web sites. In fact, the more, the better. Get maximum exposure!*

Approach 1 is a **PASSIVE** search. Your resume is a piece of bait waiting for somebody (e.g., a recruiter going through posted resumes) to get interested enough to contact you. Once your resume is posted, it works for you 24/7 while you sleep, eat, play Wii, and waste your life in other pleasant ways.

Approach 2 is an **ACTIVE** search. You promote yourself by actively sending your resume

-as a response to a concrete posting about an open position. You can find out about open positions by

- a. browsing posts on The Big Four
- b. visiting the Web site of a software company
- c. visiting the Web site of a recruiting agency

-as an offer of your testing services to employers/recruiters that didn't post matching positions, but **might*** be interested.

*As we already discussed, **every software company** is interested in motivated, hardworking, entry-level testers.

Brain Positioning

Job hunting is a JOB. You cannot perceive job hunting as a pleasant stroll in Oak Valley with your fiancé, both of you in love, with a sense of galactic harmony filling your hearts, and the future seemingly as easy as your romantic talk. With job hunting for a first job, it's just the opposite, my friends. Job hunting for a first job is hours and hours of hard work, where you have only one fuel to keep you going. That fuel is called HOPE. Be ready for hard work, do your part, and all good things will come to you. I've been there, and many who I know have been there too, and I must tell you that all that work is well worth it.

Let's talk about nuances.

The resume that you have is just a template, and in many cases it makes sense to adjust it for a particular position.

Example

If company's product is online payments and you have education and/or experience in finances, you'll want to modify your resume to put the emphasis on that education and/or experience in order to make your resume more attractive.

While in the majority of cases you'll be sending your main resume, a few special cases will require that you adjust your resume for a specific position.

I know several testers who have been hired for highly paid jobs just because of relevant education and/or job experience. The logic is simple: **it's much easier to learn testing techniques than quantum physics.**

So far, we've been concentrating on your resume, but there is also a very important part of your presentation called the **cover letter**.

- On the one hand, the cover letter is a tool to share some things that might be not appropriate on your resume, e.g., your personal passion and excitement about the position and/or the particular software company.

- On the other hand, the cover letter gives you an extra way to communicate your attitude.

A position can be posted by a company directly or via a recruiter. In both cases, the main point of your cover letter should be:

I'm ready to work unlimited hours.

I'm ready to work on weekends and holidays.
I'm ready to work for any amount of money.

Don't be afraid to include these three lines in *both* your resume and your cover letter. Remember, this is your main weapon!

Be prepared for the fact that 98% of all positions will NOT be for entry-level testers. Should that stop you from applying? Of course not! That's the whole idea – **for your first job, you have to pave your way in unorthodox ways to be successful**, and if some company needs testers with any level of experience, just send them your resume (with or without a cover letter, depending on situation). Even if the recruiter/company might not need you – or might have a good laugh at your resume, or just ignore it – WHO CARES? You have a brick wall in front of you, and it's up to you to break through it.

Here is another reason that 98% of the positions require proven testing experience. Today is March 8th, 2008. Let's **assume** that software testing as a profession doesn't exist. I can guarantee you that, if tomorrow somebody needs a software tester, they would specify that three years of experience is a must. Well, that was a joke, but the point is that in the majority of cases, the requirements listed in the job description are *too high*. Remember, this is just a game! So now you, too, know the rules.

Another argument for sending your resume (and, if needed, your cover letter) is the possibility that the software company is actually getting ready to hire an entry-level tester, so you'll just save them the trouble of posting and searching for the right candidate.

I'll end with a story about a frog that fell into a jar of milk. Instead of giving up and drowning, it started to move its legs and arms (or whatever you call those things for frogs). The milk eventually turned into butter, and the frog survived. The frog had no idea about the chemical aspects of turning milk into butter, but it just kept fighting.

Now you have a great advantage, because you have just learned one of the most important things about hunting for your first job: **In order to find your first job, you have to shoot at ALL targets:**

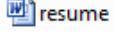
- **Invisible (passive search)**
- AND**
- **Visible (active search)**

To summarize VISIBLE targets, you will want to look for:

- Testing positions posted on recruiting Web sites, e.g., the Big Four
- Testing positions posted on software companies' Web sites in the area where you live (or where you would like to relocate)

Keep track of all the resumes you send in a separate file. You can use the table below. The last column, Documents, should have a link to the file with your resume, cover letter, and any other files you've sent to that specific recipient. You need to track whatever you've sent one way or another.

Company	Recruiter yes/no	City/ State	URL for position	Comments/ Contacts	Date of sending resume	Documents
---------	---------------------	----------------	---------------------	-----------------------	------------------------------	-----------

Google.com	no	Mount ain View, CA	http://www.google.com/support/jobs/bin/answer.py?answer=23591	My pal Robert Wang knows people at Google QA. I will talk to him. His email: bob@qatutor.com	03/08/08	 resume
------------	----	-----------------------------	---	--	----------	--

Brain Positioning

Don't be shy about promoting yourself. There is nothing to be shy about. You're not asking for some kind of super favor. This is just business. Once you are hired, not only will you benefit:

- The recruiter (if one was part of the hiring process) will get a sweet commission
- The company will get better quality software

Everyone WINS here!

So my friends, **shoot at ALL targets and GOOD LUCK!** According to my observations, approximately 300 resumes are required to get you several interviews, one of which will end up with a job offer.

And don't forget about your network. Send your resume to your friends first!

Main actions of ACTIVITY 4:

1. Send out as many resumes as possible.
2. Don't get discouraged – you will get a job.

ACTIVITY 5: Interview successfully and get a job.

Interview... This is a sweet and somewhat thrilling word for everyone who is looking for a job. It doesn't matter if it's someone's first or thirtieth interview: it's always a new, unique, and useful experience. The interview for your first job stands apart, though, because the candidate usually has zero experience in software testing as well as in what to do during the interview in order to leave a positive impression.

My purpose it to give you several good approaches to increase both your confidence and your chances to get a job offer.

Please remember and believe that if you'll work hard AND smart during the job hunting process, employers will want to meet you. Weeks and months can pass without an invitation for an interview, but one day you'll get that call or email you've been waiting for. This is not a theoretical stuff. I've been there, and I know what I'm talking about.

You'll be given an address and time for your interview. You'll be at the employer's office thirty minutes before the interview. One minute before the time of the interview, you'll take a deep breath, open the

door, and say to the receptionist, "Hello, my name is <your full name>. I'm here for an interview with <interviewer's full name>."

WAIT! What about the period of time between the invitation for an interview and the interview itself? You MUST do your homework during this time. What kind of homework? Let's elaborate:

A. Get as much information as possible about the company: product(s), business model, partners, competitors, market, history, culture, founders, etc.

The most straightforward reason for doing this is that start-up folks love their companies, and they would surely appreciate the fact that you know a lot about them.

But there is more to it.

Example

(The first "more to it")

Before her interview at one software company, my friend read that the company had a huge user base in Germany, and there were rumors about ongoing projects on product localization. Being a second generation German American, she was fluent in German, and she made sure to mention this during her interview. She was hired, and as she found out later, she was chosen over other candidates just because of her German language abilities. BTW, "Can speak German" wasn't in the job description at all. But because she had done her homework, she knew where to put an accent during the interview (and in the resume) to make herself more attractive as a candidate for that job.

Example

(The second "more to it")

My friend had an interview at one extremely popular file-sharing service. He did his research before the interview, and he found out that major music labels were likely to file a lawsuit against that company. My friend asked the interviewers, "Are you working towards an agreement with those companies that might sue you? What is your Plan 'B' if a lawsuit is filed?" After receiving a "No" to first question and "There is no Plan 'B'" to the second question, he decided that he'd better look elsewhere. As it turned out, the company was sued and shut down in several months. By doing his homework, my friend asked intelligent questions and made the right decision to avoid working there.

B. Involve your network.

For example, you can use LinkedIn to find out how people working for a particular company are connected to you. Make calls, send emails, meet for lunch; the world is small, and sometimes life brings us wonderful gifts, such as the example below.

Example

One day before an interview, a friend of mine found out that the husband of her long-forgotten cousin was the vice president of that very company. So my friend promptly arrived for dinner with a bottle of his favorite cognac, and the interview was done in a friendly atmosphere enhanced by fine alcohol. My friend got a job offer at exactly 2:00 a.m., right after the bottle was finished and the taxi driver was waking up the neighborhood with his impatient honking.

This happened in Russia, where we have our own peculiarities, but you get the point: your network can give you an easy ticket to employment.

C. If possible, try to actually use the company's product(s).

For example, if I scheduled a candidate to come to our ShareLane for an interview, I'd recommend buying at least one book from our Web site. Why? Simply to get a practical knowledge about our product. It's one thing is to tell an interviewer, "Your product is great," and quite another thing to say, "I personally like your product because <list number of concrete reasons why YOU like it>."

Example

Before my interview at one mobile start-up, I found out that my cell phone didn't support video streaming. So, I did some research, found a phone that did support video streaming, purchased it, and spent an hour playing with their product. During my interviews, EVERY interviewer asked me if I used their service, and I was able share what I personally liked, as well as what I'd love to see improved. I made a good impression with my comments, but on top of that, EVERYONE was really impressed that I actually spent time and money to get a new phone JUST to be prepared for the interview. I got the job offer. *BTW, the phone was purchased with a thirty-day money-back guarantee, so I could simply return it.*

Now think about it: What kind of message did I send when I mentioned that I bought a phone in order to get ready for the interview? Simple: it was a message about my attitude!

D. Get a nice haircut; clean, good-looking clothes and shoes; and try to get some sleep.

All of these obvious things are often forgotten. The Russians say: "The first impression is about your looks; the second impression is about your words." It's great to say smart things during your interview, but how you LOOK is also very important. I read somewhere that a good chunk of an overall impression is made within the FIRST several seconds of an interaction, so looking like Indiana Jones right after he returned from the jungle doesn't really help you make a good impression. Here are some more thoughts about your appearance during an interview:

Clothes: For the interview, it's customary to wear a business suit. I personally never do it for the following reasons:

1. It's not what software engineers wear: our professional uniform is shorts, t-shirts, sweaters, and jeans. If I were a banker, I'd come to an interview looking like a banker, wearing a formal suit and shiny leather shoes.
2. If your current manager sees you wearing business suit, the first thing he or she might think is that you are being interviewed at another company. *As we'll discover later, you must get a new a job first*

before you quit your old job. I know a guy who puts on his Armani suit every time he wants to scare his boss about his leaving the company.

Cosmetics: Don't overdo it. This is a cultural thing: in the U.S., cosmetics are more lightly applied than in Europe or Asia. After all, you are applying for a software job, not a modeling position.

Perfume: Don't use it at all. There are many people with allergies. And what if your perfume reminds the interviewer of his unfaithful fiancée? Again, in the U.S., people use fragrances much more lightly.

Sleep: It might be really hard to get a good night's sleep the night before the interview, especially if it's your first interview. But you must understand that even if your clothes and haircut look great, it doesn't really help if your face is reminiscent of a dried apple and your eyes are red like on a photo when a flash was used. Besides, you'll think more clearly if you are well rested.

To conclude this section on appearance, I can't help mentioning a funny situation that occurred when one dude came to an 11:00 a.m. interview...drunk. We talked to him and were really impressed with his expertise; he was GOOD! But, come on, if he came drunk to the interview, i.e., the place where everybody does his or her best to make a good impression, what could we expect from him in his "normal" state: dancing naked with a bottle of Stoli during his lunch break?

BTW

In many cases, you'll have

1. a phone screening
AND/OR
2. a phone interview

before the actual invitation to a person-to-person interview.

A **phone screening** is usually conducted by someone in HR (Human Resources). The QA manager gives HR a list of questions, along with a bunch of resumes. The HR person calls candidates and writes down their responses, along with any remarks about his or her impressions, on the back of the resumes. The resumes are returned to the QA manager, he or she checks the answers AND remarks, and calls (or emails) the candidate(s) who seem most promising for the position to arrange a phone interview or to invite candidate(s) for a person-to-person interview. Try to impress the HR person: **Be confident, polite, speak clearly, and put positive energy into your voice. Let your attitude be heard in every word you say!** You can make mistakes answering technical questions, but that is forgivable (this is your first job), but if the HR person doesn't like you, you'll probably never get a call from the QA manager. The bottom line: Try to make the HR person like you.

The phone interview is usually conducted by the QA manager. All recommendations about your behavior during the phone screening apply here.

What is the main difference between a phone screening (PS) and a phone interview (PI)?

1. The PS serves as the first filtering stage to eliminate candidates who CLEARLY are not a good fit, while the PI is a **real interview** that goes into depth about your knowledge/personality.

2. The PS is usually conducted by an HR person, who just goes through the questions and usually has no idea what the right answers are, while the PI is usually conducted by a professional tester or QA manager who knows the correct answers and can ask follow up questions.

During both the PS and PI, learn to use the answering techniques that we will cover next.

Okay, let's assume that you've gotten a call/email with an invitation to a person-to-person interview, you've done your homework, and you're ready to kick ass. Below are my recommendations:

1. Arrive on time.

Traffic jams, an incidental encounter with an ex-girlfriend or boyfriend, an upset stomach after burned toast, a missed train, a broken alarm clock, a flat tire, contact with a UFO – none of these important moments of your preinterview existence have any importance whatsoever to an interviewer. Try to arrive thirty minutes early to the interview. Find the right door, and then go outside and walk around the building a few times, asking God to help you get that job. At the exact time of your interview, knock at the door and tell the receptionist your name and the name of your interviewer.

2. Have a firm handshake and look directly into the interviewer's eyes.

You have nothing to be shy about. You didn't come to solicit a free meal; you've come to offer your help. The interviewer is just another human being – no better and no worse than you – and it's likely that she herself was interviewed and hired just days ago. This is just business: if you like each other you might work together; if not, you'll find an even better company across the street.

3. Answer questions without any unnecessary details but let the interviewer know that if needed, you'll go into the details.

Example

Here is my favorite illustration about the dangers of unnecessary details:

My friend was invited for an "interview" by his fiancée's parents. They had a great dinner, and the parents' blessing of marriage seemed inevitable. So my friend relaxed and decided to fill a pause in the conversation by showing off his knowledge. He asked: "By the way, I always mix up Monet and Manet. Which of them painted 'Sunflowers'?" After that question, the pause became so heavy and dense that even the CD player stopped, expecting some ugly and exciting outcome. The silence was broken by the angry voice of my friend's loyal fiancée: "'Sunflowers' was the creation of Van Gogh!" The evening was spoiled, and that stupid incident was beginning of the end of their relationship. As it turned out later, my friend's fiancée and her family constituted a rather moronic group of individuals, but my friend learned his lesson: **Don't give unnecessary details during the interview.**

BTW

Several years ago, I visited the Metropolitan Museum of Art in New York City. Suddenly an absolutely stunning painting with sunflowers attracted my attention. I came closer and read the name of the artist: Claude Monet....

4. Be friendly, yet considerate.

Whether or not you feel comfortable during your interview, try to be friendly. Make it easy for the interviewer to like your personality, *but don't overdo it*: during an interview, there is no room for stuff like:

- Anecdotes about lovers jumping from the balcony
- Stories about crazy college parties and their aftermath
- Jokes about gender, nationality, race, or religion, etc.

A professional is not a person with merely technical skills. A professional is a person who knows what to say and when to say it.

Put yourself into the interviewer's shoes.

Example

You are conducting some productive and interesting activity, like reading reviews about the newest Lexus. Suddenly your manager comes to your desk, hands you somebody's resume, and asks you to interview that person. You reluctantly walk into the conference room after looking at the professional experience of this stranger. You try to come up with some smart questions, estimating how long the interview will take (it's almost lunchtime, and your stomach has just recalled those amazing kabobs from new eatery around the corner.)

If the candidate is friendly, considerate, polite, and warm, you'll ask your babbling stomach to shut up for half an hour, and you'll try to have a pleasant talk with this great person who wants to work unlimited hours, including weekends and holidays, for any amount of money.

5. If the interviewer wants to talk, let him or her talk.

In many cases, an interviewer will want to share something with you, such as his or her excitement about the company. Don't interrupt. Be a good listener. Sometimes you can pass an interview using nothing but word "Wow" several times as the interviewer speaks and you listen.

In general, it's a very good idea for you **to discover** a subject that the interviewer is passionate about (as a rule, that subject is the interviewer himself). This is one of the greatest selling techniques around. **When you are being interviewed, you are simply selling your future time and efforts!**

6. NEVER speak negatively about your previous or current employers. THIS IS AGAINST THE RULES.

This topic might not apply for a first job candidate, but still this is an extremely important point.

Whether you used to work as a tester for a software company, an accountant for a hardware company, or a bricklayer for a construction company, you should NEVER say anything negative about your employers. The most natural question asked during an interview is: "Why do you want to find a new job?" Even if you were **forced** to look for a new job because of the racists, sexists, bigots, perverts, drunks, porn surfers, nose pickers, and other colorful personalities who clustered at your company like bees inside a beehive, RESIST the temptation to share your heartbreak story because:

a. Interviewers don't care about your problems. If you need some compassion, tell your story to your spouse or friend.

b. You will look like a backstabber and a whiner. Both backstabbers and whiners are despised.

The greatest answer I've ever heard to that question was: "I'm looking for new challenges." (Aren't we all?) After that answer, there are no other questions about why you want to quit your present job.

When you are at the interview, the Golden Rule is this: **Either say positive things about your employers OR say absolutely nothing about them.** In the unlikely case of an interviewer who tries to provoke you to be negative, simply say: "**I never say negative things about my employers.**"

Many good engineers screw up their interviews because they share stories about their sinister employers.

Complaining about your present and/or former employer(s) is called negativity. Negativity is an opportunity killer. NEVER be negative at an interview.

7. Always remember that during the interview, the interviewer is analyzing you as a potential coworker.

Ask yourself honestly whom you would rather work with:

- Someone who is an amazing specialist, but stinks like he or she spent the night in a pigsty, treats everybody as stupid, inferior beings, talks about you behind your back, and reads German philosophers merely to snobbishly mention it during a conversation

OR

- Someone who is an absolute beginner, but who is smart, motivated, honest, and positive; who wears clean clothes; who will stay at the office as long as needed to provide results and make his or her team look good; someone who will always acknowledge his or her fault and never stab you in the back

You get the point. On the surface, it might seem like that the interviewer who chooses the second candidate cares more about his or her own interests than the interests of his employer, but in reality, the second candidate will mostly likely bring much more to the company. Why? ATTITUDE is precious – the second candidate will bring comfort, reliability, and a joyful atmosphere to your team, and **team productiveness has a direct correlation on how team members interact with each other and feel about each other.**

I would sacrifice my own personal time just to get the second candidate up to speed. When I was just beginning my testing career, I met several people who helped me just because they felt my attitude. My prayers will always be with them.

Here is another note about personality – more precisely, the element of personality known as "**enthusiasm.**"

For many start-up employees, working for a start-up is not just a way to receive biweekly paycheck, but it's also a way to live a dream (whether this dream is about making millions of dollars, making a difference, or something else). **During the interview, don't be shy – show your sincere excitement about the company.** For example, let's say you are a huge fan of the baseball team, the San Francisco Giants. One person, Rajiv, is a huge Giants fan, just like you; another person, Ron, likes baseball, but it doesn't really matter to him who plays – he just likes to sip his drink and watch the game. If you have an extra ticket for a SF Giants/Milwaukee Brewers game, who would you ask to join you? Most likely you'll invite Rajiv, because you both share the same PASSION. By analogy, **startup employees are (as a rule) huge fans of their companies, and if you show that you are a fan too at the interview, you'll have much more of a chance to be invited to a game, i.e., to be hired.**

Brain positioning

If you find out that the employees of a start-up are not passionate about their company, e.g. they are not using the company's product for themselves (assuming that product is something for regular people, e.g., social networking), you should perceive this as a BIG warning sign about that start-up's future success.

In ALL successful companies that I've known (PayPal, eBay, Google, YouTube, Facebook, LinkedIn), the early employees were CRAZY about their products. I can hardly imagine a PayPalian who wouldn't use PayPal on a regular basis or wouldn't promote it to all his or her friends and relatives.

On the other hand, I've known folks with mentality: "This is a just a job" - no wonder that their start-ups ended up in oblivion. Of course, a start-up with passionate employees can end up in oblivion too, but I can hardly imagine successful young business driven by paycheck-to-paycheck attitude.

Personally, I wouldn't become an employee of a start-up where employees are not passionate about their product.

8. Honesty and sincerity win hearts. Lies and attempts to conceal something are sure ways to ruin your interview.

You simply cannot know all the correct answers to all the interview questions, because this is your first job. Just try to come to terms with that truth. So how should you handle it if you are asked a question you don't know how to answer?

DO NOT:

- wrinkle your forehead,
- move your lips,

- direct your eyes to the upper-right position (this is a sign that you are inventing something),
- do anything else that attempts to show that you are digging something from the caves of your memory

By doing any of the above, you simply insult the other person's intelligence and waste his or her time.

DO:

Say two things sincerely and confidently:

1. "**I don't know**" (this is a testimony to your honesty).
2. "**If needed, I'll learn it**" (this is a testimony to your attitude).

The combination of these two phrases is probably the most powerful tool you have during your first interview. In fact, if you asked me to share the recipe for a successful interview, I'd give it to you in one sentence: "**Demonstrate your attitude, and if you don't know something, say, 'I don't know, but if needed, I'll learn it.'**" This might sound like oversimplification, but it works. Check it out for yourself.

Even having plenty of experience, I still use this "I don't know" – "If needed, I'll learn it" combo. I'm not uncomfortable admitting this, because the world of technology is so diverse and complex that no one can know it all.

Example

During a phone interview for one of my consulting jobs, I could answer only three out of seven questions. Why? Because the four unanswered questions were about a technology I have no experience with.

Logical question: Why the heck did you apply in the first place?

Logical answer: I applied for the job because I knew the job required my knowledge of testing, not necessarily a knowledge of that particular technology.

I used the magical "I don't know" – "If needed, I'll learn it" technique, and I explained to the interviewer that his start-up needed good grey box testing. White box testing which required good knowledge of that particular technology would be needed in the future, if at all. The contract was mine.

One last aspect that I want to mention is that your readiness to learn new things is one of the greatest assets a software engineer can possess. So make sure that the interviewer knows that you are open to learn.

9. Don't get upset or angry if the interview doesn't go smoothly. No matter what happens, TRY to stay calm, friendly, and respectful.

Example

During one of my early interviews, I almost bolted from the conference room when interviewer went to get water for us. I felt terrible because interviewer could barely understand my accent*, my answers sucked, and I was very anxious. My temptation to run away was so strong that it was nothing less than Divine Intervention that made me stay in my chair. But I finished the interview and talked to several other people following it. The next day, to my enormous surprise, I got a job offer.

* To give you an idea about my accent back then, I want to present you with my dialog at MacDonald's. It was written down and preserved by my friend as a classic.

- Number three, please.
- Excuse me?
- Can I have number three, please?
- Which number?
- Number three.
- Three?
- Yes, three.
- Got it. It was three, right?

The thing here is that **it doesn't matter what YOU think about how interview is going**. What matters is **what the interviewers decide between themselves after the interview**. So try to relax and do your best. It's easier to say, "Try to relax," than to actually relax, but it is doable. The best techniques I've found are:

1. **Decrease the value of this particular job** by thoughts like, "Is this the only company in the world?" or "This company is too far from my home," or "Maybe I can get a better salary at another company," etc.
2. **Try to image the worst thing that could happen**. Are you going to be put in the electric chair or be eaten alive by a great white shark if you don't pass that interview? No. If you don't pass it, you'll just get upset, get over it, and then send out more resumes and get another interview at an even better company.

10. Never cancel an interview until you accept a job offer.

Example

One friend of mine had very smooth interview at Company A. He felt loved and was expecting a job offer any minute. So he was careless enough to cancel an interview at Company B. Guess what: the folks from Company A never called him back. He waited for a week and then called them himself. The QA manager from Company A apologized for not calling but told my friend they decided to hire another candidate. My friend rushed to call Company B, only to find out that no one was interested in talking to him anymore because the position had already been filled.

But canceling interviews has even more negative sides. By canceling an interview, you deprive yourself of the valuable opportunity to:

- Find out what questions are asked at interviews

- in general
 - at the company with a certain type of technology
 - in a certain geographical area;
-
- Learn the correct answers to the above questions
 - Polish your communication skills
 - Improve your general interview skills

Interviewing well is a separate skill. It might sound like a paradox, but *in reality there is very little relationship between how good a person looks during the interview and how well that person would work.* There are lousy workers who pass an interview with flying colors, and there are great specialists who barely make it to a job offer. What is required to improve any skill? Simple: you need a lot of practice! With each new interview, you'll learn more and thus improve your interview skills. Those skills will help you to find your first job, and they will continue to be great professional assets that you'll use hunting for each new job!

11. Remember that an interview is a DIALOG, not an interrogation.

A friend once told me that the night before her first interview she had a dream that her interview was just like a scene from the movie *Conspiracy Theory*, when Mel Gibson was tied to the chair, a bright light shining into his eyes, his eyelids taped open, and the iron voice of the evil daddy asking scary questions.

Your imagination can create the most fearsome situations, but the truth is that nobody is going to interrogate you during a job interview. You are not going to be tortured, made miserable, or be given a choice: "Death or the right answer to what a test plan is."

An interview is simply a conversation. As a rule, it's the type of dialog where one person asks questions and another person answers those questions, but in many cases, it's a great idea to turn the interview into a typical conversation where people ask each other questions. An interview is a great source of knowledge, so use that opportunity to learn things. Usually at the end of an interview, it's customary for the interviewer to ask you if you have any questions. But my point is that you don't have to wait for that moment. Try to elegantly (elegance is key here) direct the conversation into the two-way mode. For example:

Interviewer: What experience do you have with Python?

You: I know the basics about string operations, file management, and CGI scripting. What Python skills are, in our opinion, the most valuable in your company?

If you sense that the interviewer wants to keep things in an I-ask-you-you-answer-me format, do it his or her way – **don't push!**

Also, you'll probably make it easier for the interviewer if the two of you have less formal communication. What are the benefits of having less formal communication?

You let interviewer be heard. We all love to be heard, and we like people who listen to us.

You fill the interview time with more than just questions about your skills. You don't have too many skills when you are applying for your first job.

You have more opportunities to talk about your attitude. Your attitude is your main advantage, so use every opportunity to talk about it.

You'll get a lot of information on many formal and informal subjects. A friend of mine once told me about a situation where some interviewers were talking negative about their colleagues from the same company. What kind of unhealthy team is that? My friend simply refused to work for them.

12. Use professional terms.

You've learned many new terms: *bug, front-end, database, test case*, etc. USE THEM during the interview. Subconsciously, people trust the expertise of speakers who have a professional vocabulary. It's one thing when a candidate speaks about technical stuff using common language, and it's a totally different thing when a candidate (even with ZERO experience) uses professional terms fluently. By the time you finish this Course, you'll know the definitions for the majority of professional terms you'll ever encounter during your testing career, and you will have every right to use those terms because **you know what you are talking about**.

13. Remember your mantra and make sure the interviewer knows these things about you:

"I'm ready to work unlimited hours."

"I'm ready to work on weekends and holidays."

"I'm ready to work for any amount of money."

You are probably tired of me mentioning "attitude" so many times, but please understand that I'm doing this simply to hammer this stuff into your subconscious, because **attitude is the only real asset that you can put on the table when you are looking for your first job.**

14. Below is a list of typical interview questions and my recommended answers.

Please note that your attitude should be felt in each answer!

Q. Why did you decide to become a software tester?

A. I've always enjoyed digging into the features of software to discover any problems. To me, software testing is the art of finding ways to break the software, and I want to master that art. Besides, I've always dreamed of being part of a cool Internet project. I'm confident that I have the qualities that can help me to be an excellent professional. For example, I'm detail-oriented, work well under pressure, and I'm ready to sacrifice my personal time for the job. Most of all, I feel that I can contribute to the success of your company.

Q. What do you like most about software testing?

A. On the one hand, I like the fact that testing requires intuition, creativity, and technical skills. On the other hand, I like the importance of testing, because a tester produces concrete, valuable result by helping to stop bugs from being released to end users.

Q. What are the key qualities of a good tester?

A.

- Honesty
- An “I-can-break-it” attitude
- A genuine interest in software testing and pride in doing a tester's job
- An urge for constant professional growth
- The ability to quickly switch between tasks
- An informal concern about the product
- The ability to work well with people
- The ability to remain effective and focused under pressure
- Having no doubts about sacrificing personal time when it's required by the job

I think I have all of these qualities.

Q. Tell me about your short- and long-term plans for your career in software testing.

A. I want to become an expert at my trade. My short-term plan is to find a job and apply my theoretical knowledge to actual work. My long-term plan is to constantly improve my professional skills and possibly specialize in one aspect of testing – for example, the automation of regression testing. Whatever my plans are for the future, if I am hired, my focus will be to get up to speed ASAP and go above and beyond in order to become an effective professional.

Q. In the very unlikely scenario that the company needs you to come into the office during the night, would you be willing to?

A. Yes, I'll be available 24/7. (*The key here is to give your answer quickly and in a confident voice.*)

Q. In the very unlikely scenario that the company asks you to come in during weekends and holidays, would you be willing to?

A. Yes. The job comes first. (*The key here is to give your answer quickly and in a confident voice.*)

Q. During crunch time you might need to work more than eight hours a day. Are you comfortable with that?

A. Absolutely. I know that a tester's job often requires unlimited hours. I'm ready for it! (*The key here is to give your answer quickly and in a confident voice.*)

Q. Can you work on several projects at once?

A. Yes, I can quickly switch my attention and focus between tasks.

Q. How do you deal with time pressure and pressure from the management?

A. I'm completely aware of the fact that working for a start-up is interesting and rewarding, but also involves a lot of hard work when everybody is under pressure from time, venture capitalists, competitors, etc. I'm ready, and I can work in this type of environment.

Q. Describe your relevant experience and education.

A. (The answer depends on your situation. Try to present your education/experience in a way that somehow relates to testing. Beta testing experience is a great asset in answering this question).

Q. What is your biggest professional achievement?

A. (Again, the answer depends on your situation. As a rule, this type of question is not even asked if you are applying for your first job. In any case, though, be prepared to answer this question. Each of us has something to be proud of.)

Q. Why are you leaving your current employer?

A. (As a rule, this question is not asked if you are applying for your first job. In any case, NEVER say negative things about any previous employer.) Here is the standard answer if your experience is not in software testing: I feel that software testing is a great match for me, and I'm looking for new opportunities.

Q. What are your biggest disappointments at your present position?

A. (As a rule, this question is not asked if you are applying for your first job. In any case, NEVER say negative things about any previous employer.) Here is the standard answer: I have no disappointments. My priorities are hard work and constant professional improvement at my current position. My attitude is, "What can I give to my company to make it more successful?"

Q. Would you prefer to work for a large, established company or a start-up?

Choose whatever works best in your situation:

I prefer to work for a large, established company, because all the processes are already in place, and I can get up to speed quickly.

OR

I prefer to work for a dynamic start-up, where I can participate in the creation of new processes and help to start the QA department from ground zero. How cool is that!

Q. Give me an example of a complex situation and solution that you've found.

A. (As a rule, this question is not asked if you are applying for your first job. In any case, though, be prepared to answer this question.)

Q. Tell me about your strengths and weaknesses.

A. (Be prepared to answer this question. Give a humorous example of some small weakness, e.g., "I like to play poker;" followed by something really powerful about yourself – "I'm ready to work unlimited hours.")

Q. What would you like to improve in your career, and what are you doing about it?

A. (As a rule, this question is not asked if you are applying for your first job, but be prepared to answer it anyway. For example: I want to improve my programming skills. I'm in the middle of an online course on Python.)

Q. Would you prefer to work as a member of a team or independently? Why?

A. (The answer depends on your personal preferences. Be prepared for this question).

Q. Why do you want to work for our company?

A. (Remember what you've found out about the company and passionately tell the interviewer about what attracted you about the company.)

Q. What do you know about our company? Have you ever used our product?

A. (Share your knowledge and positive experiences. DON'T CRITICIZE.)

Q. Why should we hire you over another candidate?

A. (This provocative question is designed to catch you off guard and observe your reaction, so be ready. In a confident voice, tell the interviewer what you will do if you are hired:

- I will use all my knowledge, time, and efforts to meet your expectations.
- If I don't know something, I'll learn it fast.
- This job will be my first priority in life.

15. Make a speech at the end of your interview.

At the end of your interview, conclude with a powerful bang; with all your passion, honesty, and sincerity say something like:

"I know that I don't have much experience, and I realize that there might be more qualified candidates. But please give me a chance. That's the only one thing I ask for. I'm ready for a low salary, I'm ready for hard work, I'm ready for unlimited hours, and I'm ready to learn and become productive as soon as humanly possible. Please give me that chance."

Seriously, if somebody without any experience would say this to me, I'd hire him or her (if I liked that individual). On the one hand, I'd help somebody to "learn to fish," and on the other hand, I'd feel better by the simple fact of helping. Also, I know that when you've given a chance to someone, that person is **usually** forever grateful to you. That might sound selfish, but I enjoy the fact that there are folks who have gratitude in their hearts for what I've done for them.

16. Always send a thank-you email to the interviewer after the interview.

It doesn't matter if you think the interview was successful or not – this is just good manners. And in the end, there could be a situation when, during a tie with another candidate, your thank-you email will tilt the scales in your favor.

We're finished with the interview, but before we proceed further, let me share a little more about job hunting.

One of the hardest things to deal with is rejection. Be ready, though, because most likely there will be PLENTY of:

- Explicit rejections (e.g., "We cannot hire you at this time.")
- Implicit rejections (e.g., no response from the company after you send them your resume)

It might be extremely painful, but **you have to learn to take rejections as a part of the game.**

Brain Positioning

Do you know how many rejection letters the famous American writer Jack London received before he was first published? More than **600**.... This means that more than 600 times after he sent his story to a newspaper he received a letter saying something like, "Dear Sir, we cannot accept your story for our newspaper." Here is my question to you: Do you think that 600-plus rejections indicated that Jack London's stories were worthless? I don't think so. There can be many reasons for rejection, and in many cases, the **real value is not considered** during the decision making process that leads to rejection.

So the next time you get rejected, don't think it's because you are not worthy of that job. Instead, remember Jack London and his more than 600 rejections, how he kept trying and finally became accomplished, famous and rich. The first lesson about rejections is: **Rejection is not necessarily a true indication of your abilities; therefore, rejection is not a reason to stop trying to get what you want.**

Here is another lesson about rejection: **Try NOT to take rejection as a personal insult or as a curse on your destiny.** Life constantly gives us feedback – take rejection as some feedback from life. Try to find some meaning in it.

- Maybe you NEEDED that rejection to polish your technical and/or people skills.
- Maybe you NEEDED that rejection to direct you to another much better position in another much better company.
- Maybe you NEEDED that rejection to interrupt your race for money and pay a visit to your parents.
- Maybe that rejection didn't make any sense and happened just because...well, because s^&* happens.

So don't make a personal case out of rejection – as painful as it can be, it's much more productive to cool down and analyze what (if anything) went wrong rather than to blame your destiny, or the interviewer, or whatever. Everything happens for a reason. Shake off the bad feelings about rejection

like dust from your shoes, try to understand what kind of lesson there might be for you, and then proceed with your search of happiness.

And here is my last point about rejection: *Think what the world would have lost had Jack London given up hope because of those rejections...* and think how much YOU can give to the world if you have a better life! So be brave and don't ever lose hope, because if you keep trying, one day you'll get where you want to be – no doubt about it.

Some things about the post interview:

1. After the interview, the interviewer gives his or her feedback to the hiring manager, and the hiring manager makes a decision based on the feedback from all interviewers *and* his or her own opinion. If opinions vary and/or the hiring manager is not sure about the candidate, the candidate might get an invitation for another interview.
2. Sometimes it takes a week or more between your last interview and a call from the company with a job offer. Don't lose your optimism if nobody calls you the next day after your last interview.
3. It depends on the company whether rejection comes in the form of an email, a letter, or not given at all.

Let's proceed.

Brain Positioning

When a company hires a person, TWO needs are satisfied:

1. The employee's need for employment
2. The employer's need for an employee

Each of us knows about the need to find a job, but you should know that in the case of an open position, the company has the same "I NEED" situation that you do during job hunting. That "I NEED" situation might be so stressful for the hiring manager and the team members that a job offer signed by a candidate is taken as a blessing and a relief.

Here is a story about George and Olga.

George and Olga's Story

Company N. desperately needed a software tester. In 1999, the Silicon Valley job market was so hot that it was difficult to find anyone who could tell the difference between a keyboard and a monitor, and who wasn't already employed at some software company. This being the case, George's manager told him to go to a job fair and engage in belly dancing, bribery, cheating, making empty promises – WHATEVER it took to

acquire somebody's signature on a job offer. Naturally, George knew that his chances were close to zero.

During that time, Olga was employed as a tester. Her role at the job fair was to distribute flyers with an advertisement about her company. Her energy and charisma attracted George, who had absolutely lost hope by then and just wanted to chat with a nice girl. When George found out that Olga was a tester, he experienced a short spike of hope that was immediately cut off by Olga's confident expression of loyalty to her current employer.

George decided to change his tactics. He invited Olga to a fine restaurant, where he used the time-honored approach of getting a woman drunk. Ordering drink after drink for Olga and himself, he described his company as "paradise on earth," "the most promising company," "having the coolest product", "the best team in the world," etc. Between the sixth and seventh drink, he put a job offer in front of Olga, saying in English, "You won't regret this." Olga responded in Russian: "A figli!" (which means "Why not!"), and signed the job offer.

The company she signed this job offer with became one of the most successful Internet projects in history, and Olga - who is still working there - has never regretted her decision.

Lecture Recap

1. There will always be a need for an inexpensive yet smart and motivated workforce in software testing. USE that need to get your first job!!!
2. Getting a job in software testing with zero experience is absolutely a realistic thing.
3. For your first job, your attitude is of primary importance.
4. The gist of having a right attitude is simple: GIVING.
5. Mantra of beginner tester:

"I'm willing to work unlimited hours."
"I'm willing to work on weekends and holidays."
"I'm willing to work for any amount of money."

6. Stages of job hunt:

ACTIVITY 0: Tune up your attitude.

ACTIVITY 1: Let people in your network know that you are looking for an entry-level position in testing.

ACTIVITY 2: Create a resume.

ACTIVITY 3: Find recruiting agencies that look for professionals for software companies in your target market.

ACTIVITY 4: Launch a campaign dedicated to self-promotion.

ACTIVITY 5: Interview successfully and get a job.

7. Your network can give you an easy path to employment.

8. Professional networking tools: LinkedIn and Facebook.

9. Always have your resume ready, and don't be shy about emailing it.

10. A resume is the presentation of your knowledge and your virtues.

11. One of the most challenging and important things for your first resume (for testing job) is to find previous activities that relate to testing and make an effective presentation using that info.

12. While creating a resume, use

- concrete figures to demonstrate your achievements
- powerful verbs and adjectives to express your key points

13. Wherever you work, make sure to keep records of important things you do for your company.

14. Get an experience in beta testing.

15. Think twice before putting your home phone in your resume.

16. USE SPELLCHECKER.

17. Get second opinion about your resume.

18. Use editing services of native speaker to polish your resume if you are looking for a job in a foreign country.

19. Big Four Web sites for those looking employment in the U.S.:

- CraigsList.com
- Dice.com
- HotJobs.com
- Monster.com

20. Recruiter is the best friend of a job hunter.

21. Keep checking for new job postings, and update your list of recruiters every day.

22. Looking for a job is also a JOB!

23. Two main approaches during job hunt:

- Passive search
- Active search

24. Things to do before the interview:

- A. Get as much information as possible about the company: product(s), business model, partners, competitors, market, history, culture, founders, etc.
- B. Involve your network.
- C. If possible, try to actually use the company's product(s).
- D. Get a nice haircut; clean, good-looking clothes and shoes; and try to get some sleep.

25. Things about live interview:

1. Arrive on time.
2. Have a firm handshake and look directly into the interviewer's eyes.
3. Answer questions without any unnecessary details (remember the story about "Sunflowers")
4. Be friendly, yet considerate.
5. If the interviewer wants to talk, let him or her talk.
6. NEVER speak negatively about your previous or current employers. THIS IS AGAINST THE RULES. *Negativity is an opportunity killer. NEVER be negative at an interview.*
7. Always remember that during the interview, the interviewer is analyzing you as a potential coworker.
8. Honesty and sincerity win hearts. "I don't know, but I'll learn it" is the best thing to say when you don't know the answer.
9. Don't get upset or angry if the interview doesn't go smoothly.
10. Never cancel an interview until you accept a job offer.
11. Remember that an interview is a DIALOG, not an interrogation.
12. Use professional terms.
13. Remember and promote your mantra.
14. Make a speech at the end of your interview.
15. Always send a thank-you email to the interviewer after the interview.

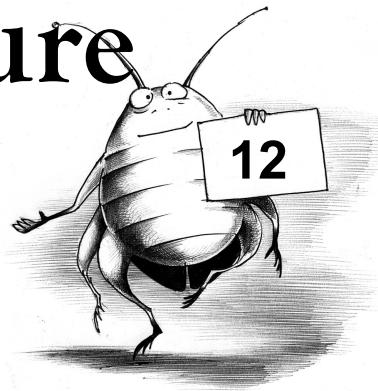
26. Employment is a conjunction of two needs: need in employee and need in employer.

Questions & Exercises

1. Start looking for a job in software testing.

GOOD LUCK!!!

Lecture



Bonus Tracks

Lecture 12. Bonus Tracks.....	339
How Much Technology Knowledge Must A Beginner Tester Have.....	340
What To Do If You Are The First Test Engineer At A Software Start-Up.....	341
What If You Are Asked To Do Test Automation As Your First Task.....	346
What Are Stock Options.....	346
How To Spot A Promising Start-up.....	349

That's thirty minutes away. I'll be there in ten.

- Mr. Wolf (movie "Pulp Fiction")

Opportunities multiply as they are seized.

- Sun Tzu

This is just a collection of short unrelated articles that may be useful for your career in testing.

How Much Technology Knowledge Must A Beginner Tester Have

Here is my position: for your first job in testing you **must**:

- Know how to use email and a Web browser
- Be familiar with black box testing techniques (Boundary Values, etc.)
- Be familiar with the main concepts of test cases, SDLC, and bug tracking

Below is a quick technology training that I recommend in addition to this Course. But this doesn't mean that you have to delay your job search*: educate yourself **while looking for a job!** And, of course, you should keep learning through your entire career. As my dear friend and tutor Nikita Toulinov says: "**If you haven't learned anything new today, your day was wasted.**"

* *I'm assuming that a job search is in your current plans.*

Here is an important thing about technology education: **Don't just read stuff – execute it!** Let your fingers remember keystrokes, and let your curiosity explore the PRACTICAL side of technology.

Example

- If you are learning Unix commands, don't just read that the command "mkdir" creates a new directory; go ahead and immediately type and execute that command and then go into that directory (the command is "cd <directory name or path to directory>").
- If you are learning about lists in Python, go ahead and create a list and start playing with it.

Remember what Confucius said: "**I hear and I forget. I see and I remember. I do and I understand.**"

Of course, you can spend YEARS learning and practicing the following things. So, we are talking about quick training to get familiarized with them. But acquired skills and knowledge will help you during your first interviews, and your resume will be more attractive if it includes a line like "Familiar with the basics of HTML, SQL, Python and Unix".

I assume that you have Windows OS installed on your computer.

1. HTML

Learn it and execute it here: <http://www.w3schools.com/html/default.asp>.

Pay special attention to elements of the Web form.

2. SQL

Learn it and execute it here: <http://www.w3schools.com/sql/default.asp>.

You can also install Base from Open Office to expand your knowledge (<http://www.openoffice.org/>). Use Menu/Insert/Query (SQL view) to interact with the DB with SQL.

Your purpose is to:

- a. Get a basic understanding of what a relational DB is
- b. Learn the basic syntax and usage of the following SQL commands:

CREATE
SELECT
UPDATE
INSERT
DELETE

3. Python

Learn it here: <http://docs.python.org/tut/tut.html>.

Download and install the Windows version from here: <http://www.python.org/download>.

Get familiarized with numbers, strings, and lists.

4. Unix

Learn it here: <http://www.ee.surrey.ac.uk/teaching/Unix>.

Download and install the Linux-like environment Cygwin from here: <http://www.cygwin.com>.

Get to know the basic concepts of Unix OS and get confident in using the basic commands.

What To Do If You Are The First Test Engineer At A Software Start-Up

Below is a list of first steps for that happy camper who is the first and only tester at a software start-up. The order of execution doesn't matter, but 1 and 2 are the most important.

- 1. Install a bug tracking system**
- 2. Learn the product, write a set of Acceptance Tests, and start executing them before each release**
- 3. Bring good standards and processes to the attention of your colleagues and management**

1. INSTALL A BUG TRACKING SYSTEM

Your purpose as a tester is to find and address bugs, so a bug tracking system is one of the most important instruments you need to do your job. When you begin working at your start-up, it's likely that:

1. The developers already use some free BTS (let's assume that it's Bugzilla) OR
2. Bugs are being tracked by something like an Excel spreadsheet

In the first case, create a Bugzilla account for yourself and ask the Bugzilla admin to

- a. Give you permission to close bugs (common users cannot do it) and
- b. Make you the default QA contact (in this case, you'll be getting emails when bugs are filed/modified/closed).

In the second case, ask the IT guy or gal to help you install Bugzilla (<http://www.bugzilla.org/>) on some server on the intranet. After Bugzilla is installed, do steps a. and b. listed above.

BTW

Bugzilla is not the most sophisticated BTS out there in terms of functionalities, but it's free, reliable, and easy to install and maintain.

The Bug Tracking Procedure (BTP) of Bugzilla is simple and very easy to understand. Spend some time playing with it: e.g., file bogus bugs and try to follow different flows of BTP. You can become a Bugzilla expert (in terms of BTP) within an hour.

One of the challenges for you will be to encourage the developers to follow the BTP, because while they tend to like the idea of Bugzilla, they tend not to like the fact that they have to use it properly -i.e., not only file bugs, but also update the Resolution and Status (or "state" in Bugzilla terms) as bugs are being repaired. Just have the patience, tact, and wisdom to communicate the necessity properly Bugzilla usage. You can find a diagram with the Bugzilla BTP here:
<http://www.bugzilla.org/docs/3.0/html/lifecycle.html>.

Another important thing is to ask your coworkers from other departments than dev and QA (i.e., marketing, product management, business development, etc.) to file bugs using Bugzilla. As a rule, these folks prefer to communicate bugs via email or a friendly chat. This is not a good practice for many reasons - for instance, the difficulty of tracking the status of a bug. So make a presentation of Bugzilla usage for your start-up brothers and sisters, and be ready to answer questions and provide Bugzilla educational support during and after your presentation.

BTW

Bugzilla actually has the functionality that allows filing bugs by sending an email to Bugzilla. This functionality is LOVED by non-technical folks—talk to your Bugzilla admin about it.

2. LEARN THE PRODUCT, CREATE A SET OF ACCEPTANCE TESTS, AND START EXECUTING THEM BEFORE EACH RELEASE

Let's assume that you have been hired to do manual functional testing.

Testing activities can be started in many different ways. Your manager might ask you to work on just one part of a Web site: e.g., start testing the Checkout. But in the majority of cases, you are simply told, "Start testing," and it will be up to you to develop the strategy. Let's talk about that strategy.

Here are the steps that I recommend:

- Learn the product
- Create a set of Acceptance Tests
- Start executing those Acceptance Tests before each release

As you've already learnt, exploration is one of the best sources of information about a Web site. If you are hired by some big corporation, you usually get a temporary mentor who helps you to get started. **In a start-up, as the first tester you'll get a lot of sympathy, but nobody will be babysitting you because everyone has more important things to do.** So it will be up to you to dig up data and start testing. Of course, you'll be asking for help, but as a rule, your first knowledge will come from exploring. Besides, once you are familiar with Web site functionalities, your questions will make much more sense to the developers. Make sure to take notes about any testing ideas that occur to you.

Let's assume that you've done enough exploring and questioning, and you are now ready to start creating the test documentation. Your first piece of documentation should be a test suite with Acceptance Tests. Why Acceptance Tests? Let's say that your Web site already has n number of important functionalities: e.g., Registration, Search, Checkout, etc. In the future, you'll need to write test cases to perform deep functional testing, digging into the nuances of each of them, but right now you don't have the time for this, especially if there are frequent releases (many start-ups have one release per week). So you have a dilemma:

- Either you write test cases in order to do a **thorough** testing of **only one or a few** chosen functionalities: i.e., go into *DEPTH*
- Or you single out **the most important flows** for **each** existing functionality and write test cases to check **all** of those flows: i.e., go into *BREADTH*

Again, each case is different, but I find it logical to go into **breadth** when you are just starting your testing.

It's usually good idea to create an Acceptance Tests in the form of an Acceptance Checklist (see downloads on qatutor.com). The idea is to have a document that can be used to check out the **general health** of the software before each release. Another important aspect of an Acceptance Tests is that it is an actual testing document; i.e., **a set of information stored, not in someone's head, but in physical form.**

Your efforts to create and execute Acceptance Tests will be largely welcomed by your coworkers, because by executing those tests you'll be preventing a lot of nasty bugs from going into production.

3. BRING GOOD STANDARDS AND PROCESSES TO THE ATTENTION OF YOUR COLLEAGUES AND MANAGEMENT

In terms of politics, this is the hardest part of all for both the beginner and the experienced tester. Start-ups are famous for their atmosphere of democracy and openness, and even a slight attempt to put some limitations on those great qualities can meet with opposition—not because of the nature of the limitation, but because for many start-up employees the word "process" is associated with evil. But even in start-ups, we DO need good processes and standards (further P&S). The reason is simple: good P&S make our work more efficient and easier to perform. The very important prerequisite of a good P&S is: "**The only reason a P&S should be introduced is to improve things.**" This motto sounds very natural, but you'd be surprised how many P&Ss are introduced for other reasons, like "At my old company we did it that way."

Now, how can a P&S be introduced in the first place? When you come up with new ideas, especially about new processes, the smart way to do it is to collect feedback first and then make your idea public. The reason is that start-up folks are usually very smart, creative, and active. So, if your idea comes out of the blue during an engineering meeting, you'll probably get a lot of opposition; there's nothing personal about it—your colleagues will just try to express their opinions. Instead, let them do it **before** the meeting: *meet privately with some of the influential folks within your company (PMs, developers, managers), share your idea, ask them for feedback, and try to incorporate their opinions into your future presentation.* This way, when you go public with that presentation, many respected folks will already know about your idea and support it. Another approach, which can be combined with the one above, is to ask one of your more senior colleagues to help present your idea in public.

It's different in every company, but usually **a quality related R&S is considered to be officially accepted once it's been voted for at the engineering meeting.**

The first pieces of a P&S that you can gradually, gently, and smartly introduce are:

- a. Bug Priority Definitions
- b. Bug Tracking Procedure
- c. Bug Resolution Times

a. Bug Priority Definitions

Bug Priority Definitions is basically internal standard (within a company) about what to call P1, P2, P3, and P4 bugs. Definitions create a guide to help assign correct priority for each particular bug. There WILL be room for dispute (about the priority of some bugs) even after the Bug Priority Definitions are accepted, but in the many cases you can simply point to this document and say: "*We've all agreed on our Bug Priority Definitions, and this particular bug matches the criteria for P2.*"

b. Bug Tracking Procedure

The Bug Tracking Procedure (BTP) is set of rules about how bug tracking should function from the moment a bug is found and filed (or re-opened) to the moment when a bug is closed. Everyone who uses the BTS for more than simple bug filing should understand the BTP. The following is an illustration of the importance of proper BTP use.

Example

There is a P1 bug that stops the release (a showstopper). At 8:00 pm on Friday, the development manager checks the BTS and finds out that the bug wasn't fixed. The responsible programmer, who is also the only person capable of fixing that bug, has already left the office in stealth mode and doesn't pick up his phone. So the manager tells his manager: "We cannot go live; we'll have to wait till Monday."

In this situation, everyone suffers:

- the programmer, who didn't fix the bug
- the manager whose people don't care
- and the rest of the company, because the release doesn't take place

But to make matters uglier: in reality, the bug WAS fixed, but the programmer didn't change its Resolution to "Fixed"! So, not taking a simple fifteen seconds of action—changing the bug Resolution—stopped the release from going out. This happened because the programmer

- either didn't know about the accepted BTP at his company
- or negligently ignored it.

The above example is an extreme case, but in reality, not knowing about or ignoring the BTP by those who should know and follow it brings A LOT of inconvenience, miscommunication, and wasted energy.

BTW

If you have Bugzilla as your BTS

- you can use the default Bugzilla BTP (see link above) or
- you can develop some kind of custom process, but this will require a Bugzilla code change, and I don't think you need that.

If you use commercial BTS software like Test Director (Mercury Interactive), it usually comes with a default process with the possibility for easy process customization (no BTS code rewriting is needed).

c. Bug Resolution Times

Why do we need bug priorities? Is it only about making some bugs more important than others? Let's say that bug #234 has priority P1 and bug #211 has priority P2. Naturally, #234 should be fixed before #211, but here is the question: What is the time frame for fixing that P1 bug #234? Is it two hours, three days, or one week? The point is this: **bug priority doesn't make much sense if there's no concrete bug fix time frame associated with each bug priority**. The Bug Resolution Times document is where these associations are made.

Brain positioning

If there is no Bug Resolution Times standard in the company and developers don't fix their bugs within a reasonable (whatever it is!) period of time, the test engineer can be easily blamed for not putting

enough pressure on them or for not effectively "selling" bugs to them. But if that standard IS in place, it becomes the problem of each particular developer who doesn't comply; testers cannot be blamed anymore. If the company has a standard, and the developers don't follow it, it becomes the development manager's problem, not the test engineer's. That's how the P&S take care of the situation and unloads unnecessary problems from the shoulders of the testers.

What If You Are Asked to Do Test Automation as Your First Task

If you're the first test engineer at an Internet start-up, chances are that the first thing you'll be asked to do is ... test automation. Why? There are many reasons, but the most common one is that it's typical for non-QA people to think that TA is the most important part of the testing activities.

Example

When I joined a certain start-up as its first tester, my first assignment was to do load/performance testing. I was in total despair because "load/performance testing" sounded to me like "climb Mt. Everest" - very cool and almost impossible to do. I called my mentor, and he said:

"You've been hired for manual functional testing, and you were told that this was what this company needs. Communicate with your manager about testing priorities."

So I asked my manager to give me some time to explore the functionalities of the Web site, and I also asked him if load testing was really more important than functional testing.

He said, **"No, we need to release the product ASAP. That's why we hired you."**

I asked, "What about load testing?"

He replied, "Oh, that. It's great to have it."

I replied, "Let me start with functional testing and later, if necessary, I'll learn the load/performance testing techniques."

He agreed. So I started to do what I was hired to do, and we revisited our conversation about load/performance testing ten months later—after we hired four more testers for our QA department.

There are two things to understand here.

1. In 99% of all cases, initially (in regard to test activities) the company needs manual functional testing.
2. If you are hired as black box tester, then your manager cannot expect you to be immediately qualified to do test automation. *Please note that I'm not referring to cases where a tester is hired specifically for an automation project.*

Talk to your manager about 2 things above. There is nothing to be shy about in this situation. It's all about business. The general perception that test automation is "kinda cool" doesn't make test automation more important than old good black box testing.

What Are Stock Options

Please note that I'm just giving you a GENERAL IDEA about employee stock options in the United States. Educate yourself and talk to your lawyer and tax advisor. Seriously.

Definition: **An employee stock option** is the **right to buy a company stock (also called a share) at a fixed price under certain circumstances.**

Example

Here is the most obvious way you can make money with employee stock options (let's just call them "stock options"):

- Let's say you receive a stock options grant to buy 5000 stocks at \$3 each.
- Four years after you joined the company, it went public.
- Today your company stock is being traded at \$17 per share.
- You log in into your online brokerage account and do an exercise-and-sell transaction when the expense of buying shares (\$15,000) is covered by the proceeds from the sale of 5,000 shares at \$17 apiece (\$85,000):

$$\text{Your expenses} = \$15,000 \text{ (5,000 shares x \$3)}$$

$$\text{Your pretax income} = \$70,000 \text{ (\$85,000 - \$15,000)}$$

That 70K is a gift from your company to you. In this example, you don't have to invest a cent of your own money. How cool is that!

Brain Positioning

The initial employee stock options grant refers to the stock options given to you when you become a permanent employee. This grant is negotiable before you put your signature on the job offer. When fortunes are made by regular software engineers, they are usually made with an initial stock options grant.

The fixed price of the stock option (\$3) is called the **exercise price**.

What about "certain circumstances"? Let's look at the **vesting schedule**.

The vesting schedule is a set of rules and conditions under which your stock options become exercisable; i.e., eligible to be converted into the shares.

Example

Your company gives you 9600 stock options as an initial stock options grant with a vesting schedule over 4 years. There are 48 months in 4 years, and you might expect 200 stock options per month (9600/48) to be potentially vested. But it doesn't work this way. Instead:

- 25% of your 9600 stock options (2400) are vested in a lump sum after one year of being an employee
- 200 options are vested each month over the next three years of your employment

If you quit

- before the end of one year, you'll have no stock options to exercise
- after one year, you can exercise only 2400 stock options
- after one year and one month, you can exercise 2600 stock options

If you don't quit, in four years you will be **fully vested**.

But there is much more to the "certain circumstances": expiration of the stock options grant, blackout periods, various restrictions to trading, etc. My advice to you is this:

1. Understand whatever you sign (stock options grant document, job offer, etc.).
2. Keep yourself informed about your stock options and related matters.
3. Heavily educate yourself on stock option related matters.
4. Talk to professionals (lawyers and accountants).

This all sounds obvious, doesn't it? Well, we all know that the history is full of examples when people get in trouble because of not doing obvious things.

Example

During one seminar on stock options, the lecturer told a story about a woman who lost about \$120,000 by not doing an obvious thing like paying attention to the stock options grant document.

Here's what happened. The stock options grant expires

- several years after it was granted or
- several months after an employee leaves the company (whichever happens first)

So, that woman (let's call her "Erin") left the company for another job on May 2, 2007. According to her stock options grant document she had 90 days after she left the company to exercise her stock options. After 90 days the options expire; e.g., turn into nothing. Having all the signed documents, the use of Google search engine, and lots of financially savvy friends, **Erin didn't know about that 90 days condition**. She was happy that the company stock grew like crazy, and she decided to convert her stock options into shares once the stock price reached \$100 a share. On August 15th, after the stock finally hit \$100, Erin logged into her brokerage account and was devastated by the simple word to the right of her stock options grant ID: "**Expired**." THAT must have hurt! **And it hurts even more if you realize that the situation could have been prevented just by paying attention to the document related to that huge amount of money.**

BTW

In our examples, we've been looking at the situation when a company is **public**. What if it's not, i.e., a company is **private**? Then the same rules about "certain circumstances" apply, plus there is the complication that you have to spend your own money to buy company shares: e.g., using figures from

our first example, this would be \$15,000. Now, what if you are going to leave a company that has not gone public where you have some stock options vested:

- On the one hand, your stock options will expire if you don't exercise them, and you can lose potential profits.
- On the other hand, what if you invest your money and the start-up eventually dies in peace (along with your hopes to get nice return on your investment)?

There is no universal advice about right actions in this case. It all depends on your financial situation, the economy, how much you believe in the company, etc. In many cases, this is a HARD choice to make.

Now, let's talk about acceleration of vesting. **Acceleration of Vesting (AoV) is a time machine where you put your vesting schedule.** Acceleration of vesting is triggered by certain events. The most common case that triggers acceleration is "**a change in control of the company.**" When someone buys our start-up, there is a change of control.

Example

Let's imagine that you have a standard four-year vesting schedule with a one-year acceleration clause in case of acquisition. Your first day at work is July 15, and your company is "suddenly" acquired the next day, July 16. So what happens? **My friend, you hit a jackpot!** It means that 25% of your stock options are now exercisable.

I've heard a story about one developer who signed a job offer and immediately went to vacation. In a couple of days, his manager called him and told him the company got acquired by another company and that this developer became millionaire because 25% of his stock options got vested.

As you can see, AoV is really important and, if it's not offered, you should try to negotiate it.

BTW

Some companies don't allow any AoV, because a start-up with AoV for its employees might look less attractive to investors. The reason is that an investor might worry that after the acquisition the employees will just cash out and run from the new owners like a deer runs from a "friendly" pack of hungry wolves. Investors have good reason to worry about this. Silicon Valley remembers many stories when the new owners did nothing but a damage to the acquired company.

How To Spot A Promising Start-up

Let's be clear: "Promising" doesn't mean "guaranteed to be successful." "Promising" means that **if:**

- the company will keep its pace and popularity
- the founders are not only programming-smart but also business-smart
- no start-up killing legal matter emerges (e.g., Napster's case)
- add another fifty "ifs"

then **MAYBE** this company will make some folks really rich. Maybe you'll be one of them.

So, how do you identify a promising start-up?

Let me ask you this:

- Did you ever use the Google search before Google went public?
- Did you watch the videos of lonelygirl15 on YouTube before YouTube was acquired by Google?
- Did you ever hang out on MySpace before it was acquired by News Corp?

I bet you (or some of your friends) HAVE!

What do all these companies have in common? **Their products are INSANELY popular.**

So the answer is simple:

If a start-up is insanely popular, then it's a promising start-up.

Here is some historic data about the indicators of insane popularity. Please note that at the end April of 2007 there were almost 114 million Web sites (according to netcraft.com).

start-up	Traffic Ranking*	Other indicators*	Source
Google	5	Most popular search engine	Alexa.com Wikipedia.com
YouTube	6	100 million videos viewed every day, with 65,000 new videos uploaded daily	Alexa.com Google Press Center
MySpace	Top 30	8% of all ads on the Web were on MySpace 27 million unique users per month** 350,000 bands and artists	Alexa.com News Corporation press release

* before or at the moment of acquisition/IPO

**data for all Web sites of Intermix Network (the owner of MySpace.com)

What other factors are involved?

Well, sometimes a company is kind of ... "pregnant." When a woman is pregnant, she is expected to deliver a child; by analogy, when a company is pregnant, it's expected to deliver some gold to **the folks with stocks**.

The tricky question is how to know if a start-up is "pregnant."

Money is made through

- the private sale of the company (YouTube) or
- the IPO releasing company stock to the public (Google).

Private sale: Only insiders know for sure if and when a company will be bought or sold, and on what conditions. But there are some signs that outsiders can look for, like excitement about certain technologies (e.g., VoIP - Voice over Internet Protocol) with the big guys like Google or Yahoo shopping around.

IPO: A company goes public, i.e., starts to publicly trade its stocks through its IPO (Initial Public Offering). To make an IPO happen, the company must go through a number of formalities, like filing papers with the U.S. SEC (Securities and Exchange Commission). After all the papers are filed, but before the stock is publicly traded, the company is in pre-IPO stage.

Joining a company at pre-IPO stage is a smart thing, but you should remember that **as a rule, the more confident the general public is about the pregnancy of the company, the less stock options you'll be offered as an initial grant.**

You can get a lot of information about “pregnancies” from public sources like news and articles. Stay informed! A very good source of tech rumors and news is Michael Arrington's techcrunch.com.

Joining even a pregnant start-up with insane popularity cannot guarantee that you'll make millions. I can give you a little example:

Example

Imagine that you are a founder of a start-up and your stake in the company is 60%. Your company has millions of users, good publicity, and the prospect of enormous growth. One day you are approached by someone from a really big company who offers you \$100 million for your company.

Will you be able to resist pocketing \$60 million now?

- on the one hand, you have a **real** \$60 million offered to you
- on the other hand, you have a company that in the future **theoretically** can bring you much more than \$60 million.

It's indeed a tough choice to make.

Another question: **if you are offered \$60 million, will you be thinking about the many employees of yours who have 0.01% or less in your company?**

First, let's calculate: 0.01% of \$100 million comes to \$100K. \$100K is around \$60K after taxes, and an employee would have to work for four years to get it, so that employee with 0.01% will get an extra \$15K a year.

Don't take this wrong. An extra \$15K is good money, but it's a joke compared to millions made by the employees of companies that don't sell themselves cheap. (I'll leave it up to you to answer the question, "Will you care about your employees?")

Again: stay informed. And if you are employed at a promising start-up, **keep your eyes and ears open.**

I must add that I am impressed by the Google founders who took really good care of their employees. **In August 2004, about 50% of all Google employees became millionaires, with the average value of the stock options being \$4.2 million per employee!** That statistic doesn't account for the enormous value of stocks held by Google's top officers and founders (source: salary.com).

In August 2004, it was reported that founders Sergey Brin and Larry Page, along with CEO Eric Schmidt, requested that their base salary be cut to \$1.00 (Wikipedia). Of course, they are multibillionaires, but that gesture says a lot about their decency, honesty, and their faith in the company, especially in light of recent publicity about corporate corruption in US.

So, two more factors for you to consider are:

1. The personalities and past histories of the founders (and top officers)
2. How well existing employees are treated by the company

Another important thing to look at is **which venture capital firm has funded the start-up**. If it's the top tier, like Menlo Ventures or Sequoia Capital, it means that the best brains in the VC world have put their hopes on the company. These guys only invest their money if they have good reason to expect hundreds of percent of potential returns.

As you can see, there are many ifs and zero guarantees here. One could write a book about the possibilities of start-up employees being screwed. I think that **the best-case scenario is working for a promising start-up in whose product you PERSONALLY believe**. In that case, **your reward is in every working day**.

If your start-up makes you rich, you are a double winner. If it doesn't, you are still a winner, because **working for the idea in which you believe is living a DREAM**.

THE END



Afterword

The policy of being too cautious is the greatest risk of all.
- Jawaharlal Nehru

**It's not the size of the dog in the fight,
it's the size of the fight in the dog.**
- Mark Twain

My dear friends,

our course is over, but I believe that this is just the beginning of your rewarding journey into the land of software testing. My purpose wasn't only to teach you how to test, file bugs, or find a job. There was more to it. I tried to give you HOPE, hope that the treasure called "software testing" is available and reachable. That treasure has radically changed for the best the lives of many people, and my educational efforts had one simple goal: to show you, my students, the shortest path to a new, life-changing career.

Here is how it works: **Hope invokes dreams, and dreams call for action.** A dreamer is not a person who just says: "I have a dream." A dreamer is a person who builds imaginary castles first and then goes out there, getting his hands dirty with concrete, breaking nails against the bricks, struggling with fear, bursting through the disbelief of those around him, sacrificing comfort, making mistakes, being at the edge of giving up ... and *not* giving up. The idea is not only to be excited and inspired, the idea is to materialize your excitement and inspiration into ACTIONS. If you've thoroughly studied this course, then you are FULLY QUALIFIED to work as software testers – **there's no doubt about it**, but I know that many of you will hesitate. While hesitation is a sign of sanity, I must tell you this: **Have hesitations, but don't let them rule your lives.** As I understand it, those hesitations are rooted in two things: fear and/or laziness.

Example

Several years ago I offered to help one of my friends. She was struggling financially and I decided to teach her testing for free and help her to find a job. The job market was hot, and it was easy to find a consulting position for a beginner tester at \$35 an hour (approximately \$70K a year). At first, she was really excited, but then she said that ... it was too much money and she wanted to take slower steps. When I asked her why she **refused the opportunity**, she said that she wanted to go to college first (to get a degree in computer science), get several certifications, and *then* start looking for a job. So, while all her friends seized the opportunity and were learning technology at work *while making \$4K to 15K a month*, she was working her low paying, low self-esteem job, cherishing her plans about college and certifications and a bright future in software industry. As you already guessed, she didn't enter any

college. In fact, she did nothing to change her life. She stayed where she was 10 years ago: same lousy job, same lousy car, same lousy apartment; same lousy attitude.... People around her were making fortunes, losing them and making them again, but she kept procrastinating. IMHO, it's okay to be satisfied with little, but it's not okay to be unhappy and do nothing to change your life.

Moral: Sometimes it's fear and/or laziness that make us come up with reasons to postpone action. I don't want anyone to look back and see numerous opportunities that have not been taken because of fear and/or laziness. Again, **after you've taken this course you are fully qualified to work as a software testers.** It's not going to be easy to find your first job, but it's absolutely doable. Many of my Russian readers have done it. BTW, you have an amazing advantage: you didn't simply read about testing, you took a **practical course** and thus **acquired practical skills**.

Here is another thing that I want to warn you about. The world is full of negative, pessimistic people. It's likely that you have some of them among your friends. DO NOT allow them to screw up your plans. Once you try to change your life for the best, they will almost certainly try to interfere with their loser talk, telling you, "It's not possible," "It doesn't make any sense," "You cannot change your destiny," "You are from a poor family and have no connections," "You'll never make it," etc. We all have encountered people like this. It threatens them when somebody wants to try something new.

When somebody gives you brainwashing like this, do this simple thing: first say, "I understand your position. Thank you," and then shake off that loser's advice and calmly continue to pursue your dreams. Among those negative souls there can be people who are really smart and/or who you respect, so here's the deal: Let them be smart or whatever, but don't let them influence you. **Negativity kills opportunity.** Better go ahead and clean up your social circles keeping only positive folks who inspire you, believe in you, and make you feel good about yourselves.

Remember, if you follow the advice of negative suckers and don't follow your dreams (whether they are about embarking on a new career, moving to another country, or finding a soul mate) it's YOU who will eventually regret about missed opportunities and undeveloped talents. Like Mark Twain put it:

"Twenty years from now you will be more disappointed by the things that you didn't do than by the ones you did do. So throw off the bowlines. Sail away from the safe harbor. Catch the trade winds in your sails. Explore. Dream. Discover."

Even if you don't achieve your initial target, you'll gain experience, connections, and a bunch of new ideas about where to go next! In many cases, you'll need to change your approach to reach your dream, or you might start pursuing another dream - and that's okay. Life is not static, so why should you be? While you follow your own unique path, you'll be exploring new facets of life and your personality, and every day you'll be getting a better feeling about what's right for you and what have you been born for!

Here is the last thing I want to say.

Each of us is different and life of each of us is different. That's the fact. But here is another fact: Each of us can wake up in the morning and say: **"Wherever I'm right now on this planet Earth, I'm going to do something for my Dream TODAY. Let my Dream seem ridiculous to some and unrealistic to others. But this is MY Dream, and I'm going to do something TODAY to get closer to it."**

I wish you great health, a loving family, supportive friends, fantastic careers, and a life full of beauty and inspiration.

Your friend,
Roman Savenkov

Sunnyvale, California, 2004-2008

Links & Books

1.1. SOFTWARE TESTING. Links.

http://www.kaner.com/articles.html	Collection of articles and presentations by Cem Kaner.
http://www.testingeducation.org/BBST/index.html	Video course by Cem Kaner and James Bach.
http://www.stickyminds.com	Collection of education resources

1.2. SOFTWARE TESTING. Books.

Testing Computer Software , 2 nd Edition by Cem Kaner, Jack Falk, and Hung Q. Nguyen
How to Break Web Software , Mike Andrews and James A. Whittaker
Lessons Learned in Software Testing , Cem Kaner, James Bach, and Bret Pettichord
Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional , by Rex Black
Testing Applications on the Web , Hung Q. Nguyen, Bob Johnson, Michael Hackett, and Robert Johnson

2.1. PROGRAMMING. Links.

http://docs.python.org/tut/tut.html	Tutorial on Python (you can download Python interpreter for Windows from www.python.org)
http://tryruby.hobix.com/	Interactive Ruby tutorial
http://www.cygwin.com/	Linux-like environment for Windows. Use it to learn Linux commands.
http://www.devshed.com	Huge library of free programming tutorials.
http://www.w3schools.com/html/	Tutorial on SQL. Note, that this site

	has many tutorials.
http://www.w3schools.com/sql/	Tutorial on SQL. Note, that this site has many tutorials.
http://www.ee.surrey.ac.uk/Teaching/Unix/	Nice intro to Unix

2.2. PROGRAMMING. Books.

Absolute Beginner's Guide to Databases (Absolute Beginner's Guide) , by John Petersen
Python Programming for the Absolute Beginner , by Michael Dawson
Core Python Programming , 2 nd Edition, by Wesley J. Chun
Programming Ruby: The Pragmatic Programmers' Guide , 2nd Edition, by Dave Thomas, Chad Fowler, Andy Hunt.

3.1. START-UPS. Links

http://www.paulgraham.com/startupmistakes.html	18 mistakes that kill startups
---	--------------------------------

3.2. START-UPS. Books

The PayPal Wars: Battles with eBay, the Media, the Mafia, and the Rest of Planet Earth (Hardcover) , by Eric M. Jackson
Founders at Work: Stories of Startups' Early Days (Hardcover) , by Jessica Livingston

4. CAREER AND MOTIVATIONAL LITERATURE.

Awaken the Giant Within : How to Take Immediate Control of Your Mental, Emotional, Physical and Financial Destiny! , by Anthony Robbins
The Magic of Thinking BIG , by David J. Schwartz
The Power of Positive Thinking , by Norman Vincent Peale
What Color Is Your Parachute? 2007: A Practical Manual for Job-Hunters and Career-Changers (What Color Is Your Parachute) , by Richard Nelson Bolles

Glossary

A	
Acceptance Testing	(Also called "certification testing") A quick test to determine if software is ready for release Acceptance testing follows regression testing. As a rule, test documentation for acceptance testing comes in the form of a checklist.
Actual Result	<i>In general:</i> whatever happens in reality as a consequence of something <i>Regarding the software:</i> actual output
Ad Hoc Testing	Testing done without any preparation Doing ad hoc testing, the tester just follows his or her heart to try to find bugs.
Alpha Testing	Testing done before a release When you hear "alpha testing," it refers to when the testing is done, not how the testing was done. As a rule, alpha testing is done inside the company. In some cases, alpha testing is outsourced to other companies. Also see <i>Beta Testing</i>
Also Notify (in BTS)	The list of persons who should get emails containing: <ul style="list-style-type: none">- Notification that the bug has been filed- Updates about changes to the bug
Application Core	The heart of the system responsible for processing

	Also see <i>Web Site Architecture</i>
Application Version	The version of the application formatted according to the convention accepted at the company For example, at ShareLane we have this convention: <i><major release number>.<minor release number>-<build number>/<DB version number></i> e.g., 1.0-23/34
ASCII	"Character encoding based on the English alphabet. ASCII codes represent text in computers, communications equipment, and other devices that work with text." (Wikipedia)
Assigned by (in BTS)	Alias of the person who assigned the bug to its current owner
Assigned to (in BTS)	Alias of the current bug owner
Attachment (in BTS)	File (usually an image) uploaded as an illustration to the bug
Automated Testing	Testing completely done by running test automation tools. Simple to perform yet valuable, automated testing can be done with link checkers; e.g., Xenu Link Sleuth (you can download it from QATutor.com). Also see <i>Manual Testing, Semi-Automated Testing, Test Automation</i>
Automation Script	(In the context of this course) Test automation written for the regression testing of - components - end-to-end flows Also see <i>Helpers</i>
B	
Back End	Software and data hidden from the user; e.g., Web server, application core, the DB, log files, etc.

	<p>Compared to a car, the back end pieces are items like the engine and the electrical circuit of the car.</p> <p>Also see <i>Front End</i></p>
Beta Testing	<p>Testing done after a release to a narrow audience</p> <p>The value of beta testing is allowing some representatives of our target audience to try the product and give us feedback before we release it in the open.</p> <p>Also see <i>Alpha Testing</i></p>
Black Box Testing	<p>Manual or semi-automated testing that occurs when:</p> <ul style="list-style-type: none"> - The tester usually has no idea about the internals of the back end - Ideas for testing come from expected patterns of user behavior <p>Also see <i>Grey Box Testing; White Box Testing</i></p>
Black Box Testing Methodology	A set of black box testing techniques and approaches
Boundary Testing	Testing using the <i>Boundary Values</i> technique
Boundary Values	<p>Two meanings:</p> <ol style="list-style-type: none"> 1. Extreme border values of the equivalent class 2. Black box testing technique that involves the boundary values of the equivalent classes
Brain Positioning	The most vital fundamental concepts and attitudes regarding the subject of study
Bug	<p>Depending on the context:</p> <ol style="list-style-type: none"> 1. The deviation of an actual result from an expected result 2. A problem report with the "Bug" type inside the Bug Tracking System <p>Also see, <i>Software Bug</i></p>
Bug Attributes (in BTS)	<p>Attributes of bug report in BTS</p> <p>-ID</p>

	<ul style="list-style-type: none">-Summary-Description-Attachment-Submitted by-Date-Assigned to-Assigned by-Verifier-Component-Found on-Version-Build-DB-Comments-Severity-Priority-Also notify-Change history-Type-Status-Resolution
Bug Fix Verification	A 2-step process: <ol style="list-style-type: none">1. Verification that the bug was really fixed2. Checking whether the bug fix has introduced other bugs <p>The bug can be closed right after Step 1 if the bug is no longer reproducible.</p>
Bug Owner(in BTS)	The person responsible for the next step in bug closure
Bug Postmortem	A meeting to discuss why bugs that went to production weren't caught during testing This meeting should be about improvements, not about blaming.
Bug Regression	See <i>Bug Fix Verification</i>
Bug Tracking Procedure (BTP)	The set of rules about how bug tracking should function from the moment a bug is found and filed (or re-opened) to the moment when a bug is closed
Bug Tracking System (BTS)	Infrastructure that enables testers to <ul style="list-style-type: none">- create,

	<ul style="list-style-type: none"> - store, - view and - modify <p>information about bugs</p>
Bugzilla	Free, popular, and easy to use BTS software
Build	A sub-version of the specific release
Build (in BTS)	<p>The build number of the application version where the bug was found</p> <p>If the application version is 1.0-23/34, the build in the BTS must be 23.</p>
Build ID	<p>Unique identifier of sub-version of the software for concrete release.</p> <p>For example:</p> <p>1.0-23</p>
Build Number	<p>Value that shows how many builds have been created for concrete release.</p> <p>Build numbering starts with 1 and increases in increments of 1 every time a new build is created</p> <p>For example:</p> <p>1.0-23 means that there were 23 new builds for release 1.0.</p>
C	
Change History (in BTS)	<p>A log of the changes that occur with a bug</p> <p>The Change History usually includes:</p> <ul style="list-style-type: none"> - Date and exact time (to the second) of the bug filing/change - Alias of the person who filed a bug/made a change - Fact that the bug was filed, or what was changed and how it was changed

Checkbox (in HTML)	<p>Used for situations where there is a need to select 1 or more options</p> <p>Unlike radio buttons, there are no relationships between checkboxes.</p> <p>For example:</p> <p>Making a sandwich</p> <table style="margin-left: 20px;"> <tr> <td><input checked="" type="checkbox"/> Bread</td> </tr> <tr> <td><input type="checkbox"/> Lettuce</td> </tr> <tr> <td><input checked="" type="checkbox"/> Cheese</td> </tr> </table>	<input checked="" type="checkbox"/> Bread	<input type="checkbox"/> Lettuce	<input checked="" type="checkbox"/> Cheese
<input checked="" type="checkbox"/> Bread				
<input type="checkbox"/> Lettuce				
<input checked="" type="checkbox"/> Cheese				
Comments (in BTS)	<p>This attribute has 2 main usages:</p> <ul style="list-style-type: none"> - Comments about the bug itself; e.g., provide more info about the steps to reproduce - Comments about bug-related actions; e.g., when a developer reassigns a bug to another developer 			
Compatibility Testing	<p>Checking the compatibility of our software with the expected hardware and/or software on a user's computer</p>			
Compiler	<p>Compilers and interpreters are special programs that translate code typed by humans ($a=2+3$) into the language understood by computers (0011010100101010101001111)</p> <p>The difference between compiler and interpreter is this:</p> <ul style="list-style-type: none"> - A compiler produces an executable file, but doesn't execute our code. Therefore, text files written in C++ must be compiled to become executable (launchable) programs. - An interpreter doesn't produce an executable file but immediately executes our code. Therefore, text files written in Python are already executable. 			
Component (in BTS)	<p>The component where a bug was found – e.g., Checkout</p>			
Component Testing	<p>Functional testing of a logical component</p> <p>Also see <i>Integration Testing</i>; <i>System Testing</i></p>			
Condition	<p>A preset or the environment</p> <p>For example, we can try to register using:</p> <ul style="list-style-type: none"> - a brand new email address 			

	<p>OR</p> <ul style="list-style-type: none"> - an email address that has already been used for registration <p>"Email is not in database" and "Email is in database" are two conditions.</p>
CVS	Free version control software
D	
Data	<p>A piece (or pieces) of information</p> <p>Data usually comes in the form of text, a file, a value of a variable, or a value inside a DB record.</p> <p>For each concrete situation, data is either:</p> <ul style="list-style-type: none"> - Valid <p>OR</p> <ul style="list-style-type: none"> - Invalid <p>For each concrete situation, Null input can be considered as either valid or invalid data.</p>
Data Integrity	<p>"Correctness, completeness, wholeness, soundness, and compliance with the intention of the creators of the data" (definition taken from TechWeb.com)</p> <p>If data integrity is seriously compromised in the test environment, testing might not make much sense until data integrity is reestablished.</p>
Database (DB)	<p>Conceptually, the DB is a set of virtual containers required to organize and store data</p> <p>The most popular DBs used in the Internet software industry are MySQL (free) and Oracle (not free).</p> <p>Also see <i>Web Site Architecture</i></p>
Database Administrator (DBA)	A professional who manages databases
Date (in BTS)	Date and time when a bug was filed

DB (in BTS)	The DB version of the environment where a bug was found If the application version is 1.0-23/34, the DB in the BTS must be 34 .
Debugging	The activity of a programmer to identify a buggy piece of code and fix it
Deprecated Code	A piece of code that has not been removed from the software, but should not be used Often, code is not removed but deprecated to make a safe transition from an old (deprecated) code to a new code. Deprecated code serves as a backup in case something goes wrong with the new code. Once the new code has been there for a while and there have been many opportunities to analyze the consequences of removing the deprecated code, we might want to get rid of it.
Description (in BTS)	A detailed description of the bug Here is the recommended format for Description: Description: <provide information about what happens> Steps to reproduce: <provide steps to reproduce the bug> Bug: <state what's wrong> Expected: <state what's expected> Also see <i>Summary</i>
Developer	A professional who writes software code
Development Environment	(Also called "playground" or "sandbox") The software/hardware combo where a developer writes and tests his or her code
Dirty List - White List	A black box testing technique that consists of 2 parts: Part 1: Brainstorming (black list) Part 2: Selection of items that came up during Part 1 (white list)
Drop-Down Menu (in HTML)	A list of values to choose from For example:

	<p>Card Type:</p> 
E	
Emergency Bug Fix (EBF)	A situation when a P1 bug is found in production and it's necessary to create a patch release ASAP
EBF procedure	Set of rules on how to proceed in case of EBF or EFR
Emergency Feature Request (EFR)	<p>A situation when a certain feature must be released ASAP; e.g., in the case of a court ruling or to comply with a new law</p> <p>For example:</p> <p>Our competitor has won a patent case, and we have to change some piece of our software ASAP to make it work in a different way.</p>
End-to-End Testing	See <i>System Testing</i>
Entry Criteria	<p>Certain conditions that allow to begin something</p> <p>For example:</p> <p>To make a phone call, you must have a working phone, a connection, and the phone number of the recipient. So we can say that the entry criteria for a phone call includes 3 conditions:</p> <ul style="list-style-type: none"> - a working phone is available - a connection is available - the phone number is known <p>Also see <i>Exit Criteria</i></p>
Equivalent Classes	<p>A set of inputs that are treated by software the same way under certain conditions</p> <p>In other words, under certain conditions, the software must apply the same logic to each element of the equivalent class.</p>

	<p>In some cases, the equivalent class can consist of only one element.</p> <p>Also see <i>Boundary Values</i></p>
Error Message	<p>A message that provides information about error(s)</p> <p>An error message is an important measure that:</p> <ul style="list-style-type: none"> - Guides users in case of mistakes <p>An error message is usually delivered via a Web page by a code that is specifically written for handling user errors.</p> <ul style="list-style-type: none"> - Gives debugging info to developers <p>An error message is usually provided by an interpreter (or a compiler), or by some logging mechanism: e.g., the Apache Web server records errors in a special error log.</p>
Error Handling	<ol style="list-style-type: none"> 1. How the system responds to errors made by users <p>An example is the way the system responds if a Web form is submitted with invalid data in a required field.</p> <ol style="list-style-type: none"> 2. How the system reacts to errors that happen when the software is running <p>For example, this error message: Test Portal>More Stuff>Python Errors>register_with_error.py provides info that the file register_with_error.py calls the undefined function get_firstpage().</p>
Exit Criteria	<p>Certain conditions that allow something to be considered finished.</p> <p>For example, lunch at a restaurant is finished when the bill is paid. The exit criteria for a meal at a restaurant is:</p> <ul style="list-style-type: none"> - The bill is paid <p>Also see <i>Entry Criteria</i></p>
Expected Pattern of User Behavior	Scenarios that we expect will be (OR are already) taking place as users use our software

Expected Result	<p><i>In general:</i> whatever is expected to happen in reality as a consequence of something <i>Regarding the software:</i> expected output</p> <p>Sources of expected results:</p> <ul style="list-style-type: none"> - Specification - Life experience - Common sense - Communication - Standards - Statistics - Valuable opinion - Others sources <p>The real challenge is to find expected results that serve as a true indicator of whether the software works or not.</p>
Exploratory Testing	<p>Exploration for the purpose of finding bugs</p> <p>Also see <i>Ad hoc testing</i></p>
Exploring	<p>Browsing through software UI to understand how things work.</p>
F	
Feature	<p>Depending on the context, the term "feature" means:</p> <ul style="list-style-type: none"> - the ability to accomplish a specific task (i.e., <i>functionality</i>) <p>OR</p> <ul style="list-style-type: none"> - a particular characteristic of the software
Feature Release	<p>see <i>Minor Release</i></p>
Flow	<p>See <i>Scenario</i></p>
Flowchart	<p>See <i>Process Flowchart</i></p>
Formal (Documented) Testing	<p>(In context of this course) Test execution with help of test documentation; e.g., test cases</p>
Found On (in BTS)	<p>The environment where a bug was found</p>

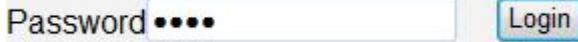
	For example: main.sharelane.com.
Front End	<p>The interface that customers can see and use; e.g., text, images, buttons, or links</p> <p>Compared to a car, the front end pieces are items like the steering wheel and the dashboard.</p> <p>Also see <i>Back End</i></p>
Functional Testing	<p>Testing with the purpose of finding bugs in functionalities of the software.</p>
Functionality	<p>The ability to accomplish a certain task</p> <p>For example, the functionality (ability) of a bottle opener is to open bottles.</p> <p>Also see <i>Feature</i></p>
G	
Grey Box testing	<p>Testing that combines the elements of black and white box testing:</p> <ul style="list-style-type: none"> - On the one hand, the tester uses black box methodology to come up with test scenarios - On the other hand, the tester possesses some knowledge about the back end, AND he or she actively uses that knowledge
H	
Helpers	<p>(In context of this course) a type of test automation needed to automate manual repetitive tasks</p> <p>The most popular example of a helper is the utility for the automated creation of new user accounts; e.g., the Account Creator used at ShareLane.</p>
I	
ID (in BTS)	The unique ID of the bug

(HTML) Image	<p>A graphic file on a Web page</p> <p>For example:</p> 
Input	<p>Depending on the context:</p> <ul style="list-style-type: none"> - data - scenario <p>Also see <i>Output</i></p>
Integration Testing	<p>Functional testing of the interaction between two or more integrated components</p> <p>Also see <i>Component Testing; System Testing</i></p>
Internet	<p>"The global system of interconnected computer networks" (shortened definition from Wikipedia)</p>
Interpreter	<p>See <i>Compiler</i></p>
Intranet	<p>Local network within a company</p>
Invalid Data	<p>Data that should NOT be able to assist in accomplishing some task; e.g., registration</p> <p>In the case of registration at ShareLane, a ZIP code must contain 5 digits. All other inputs are invalid data.</p>
J	
Job Security	<p>Ability to find a job in any economic situation. Job security is about your professionalism and not about your company</p>

L	
Legacy <something>	The word "legacy" is usually used in the terms "legacy feature" or "legacy user". It means "existing". For example, if production has a feature called "Shopping Cart", then the "Shopping Cart" is a legacy feature.
Link (in HTML)	<p>Navigation element</p> <p>The link can point to</p> <ul style="list-style-type: none">- a URL- a position on a Web page <p>Here are the typical targets for links:</p> <ol style="list-style-type: none">1. File – <i>e.g.</i>, HTML file (Web page), PDF file, TXT file, Python file, etc.2. Anchor on <i>the same</i> Web page3. Anchor on <i>a different</i> Web page4. mailto value <p>For example:</p> <p style="text-align: center;"><u>Sign up</u></p>
Linked Image (in HTML)	<p>A link presented in the form of an image</p> <p>For example:</p>  <p>The painting depicts a woman with long dark hair, wearing a red and white patterned wrap, sitting on a beach. The title of the painting, "The Moon and Sixpence", is written in green text at the bottom right of the image.</p>

Load/Performance Testing	A set of testing techniques designed to load the system or its components and then measure how the system or its components react The usual purpose of load/ performance testing is to find a bottleneck; e.g., a part of the system or its components that slows down response time.
Localization Testing	The testing used to find bugs in the adaptation of the software by users from different countries For example, if our Web site was created for an English-speaking audience and we want to localize it for a Japanese-speaking audience, we'll have to determine whether Kanji symbols can be used to create a username.
Log File	A file that keeps track of some activity There are 3 most common actions regarding data in log files: <ol style="list-style-type: none">1. Read (lines are read by humans or a program)2. Append (new lines are added under old lines, if any)3. Write (all old lines if any are purged and new lines are added)
Logical Bug	A bug in how the software processes information Also see <i>Syntax bug</i> ; <i>UI bug</i>
M	
Major Release	Major (or milestone) release that happens at the release stage of the SDLC, after the testing and bug fixes stage is over The version of a major release is presented as an integer: 7.0
Manual Testing	Testing that doesn't require any automated tools This term is usually applied towards black/grey box testing. Also see <i>Automated Testing</i> , <i>Semi-Automated Testing</i>

	A release that takes place between major releases A minor release can have one of three variants: <ul style="list-style-type: none">- Feature release- Patch release- Mixed release A FEATURE RELEASE takes place when we need to: <ul style="list-style-type: none">- Add new featuresand/or- Modify/remove existing features A PATCH RELEASE takes place when the code in production has a bug(s). In this case, we simply release the fixed code. A MIXED RELEASE is minor release with both feature related changes and bug fixes. The version of a minor release is presented as a number after the decimal point and incremented by one after each further minor release: 7.1
Mixed Minor Release	See <i>Minor Release</i>
N	
Negative Testing	Testing that checks situations involving: <ul style="list-style-type: none">- User error and/or <ul style="list-style-type: none">- System failure Also see <i>Positive Testing</i>
New Feature Testing (NFT)	The first stage of test execution where new and/or changed features are tested Also see <i>Regression Testing</i>

Null Input	No data is provided For example, if we press "Continue" on the first page of the ShareLane registration without entering anything into the "ZIP code" field, this is null input.
O	
Output	Result produced by the software in response to input
P	
Password Input Box (in HTML)	Special 1-line text box where the input is masked by asterisks or dots as the user types text Example: 
Patch Release	See <i>Minor Release</i>
PjM or Project Manager	A professional who manages projects "They have the responsibility of the planning, execution, and closing of any project" (Wikipedia). As a rule, in a start-up environment, the PjM's role is assigned to the PM.
PM or Product Manager	A professional who manages products "A product manager researches, selects, develops, and places a company's products" (Wikipedia). One of the main deliverables expected from a PM are well-written specs.

Positive Testing	<p>Testing that checks situations where:</p> <ul style="list-style-type: none">- The software is used in a normal, error-free way and/or- The system is assumed to be sound <p>Usage in a "normal, error-free way" can be defined as a scenario that accomplishes certain tasks needed to provide some value to a user. For example, registration is needed to create an account. So,</p> <ul style="list-style-type: none">- Normal usage correctly completes all steps of the registration needed to create new account.- Abnormal usage in this case would be submitting a Web form during registration where certain fields (e.g., ZIP code) have invalid data. <p>Also see <i>Negative Testing</i></p>
Postmortem	See <i>Bug Postmortem</i>
Priority (in BTS)	The magnitude of a bug's impact on the company's business
Prod	See <i>Production</i>
Production	A Web site available to our users The opposite of prod are the development and test environments where the software is being developed and tested.
Production environment	See <i>Production</i>
Programmer	See <i>Developer</i>
Process Flowchart	A graphical presentation of the process
Q	

	<p>Quality State or characteristic attributed to something (functionality, code, overall product, etc.) based on degree of match between that "something" and someone's expectations about it.</p> <p>Example #1: A tester says: "The quality of the checkout flow is good, because we fixed all the bugs." (The expectation is: "Good software is bug-free software.")</p> <p>Example #2: A user says: "The quality of the checkout flow sucks, because the UI is very misleading." (The expectation is: "Software should have an easy-to-use interface.")</p>
Quality Assurance (QA)	The set of activities targeted at bug prevention through process improvement
QA Engineer	<p><i>In theory:</i> A professional specializing purely in process improvement</p> <p><i>In reality:</i> This term is used interchangeably with "test engineer" and "tester".</p>
R	
Radio Button (in HTML)	<p>The element of a Web form that allows for the selection of one, and only one, value from a group with the same name (within the same Web form)</p> <p>For example:</p> <p>Morning plan <input checked="" type="radio"/> Go fishing <input type="radio"/> Play with kids <input type="radio"/> Repair dishwasher</p>

Regression Testing (RT)	<p>Checking to determine if legacy features are broken because of a particular change in the software; e.g.:</p> <ul style="list-style-type: none">– The introduction of new features– A bug fix <p>There are 2 reasons for regression testing:</p> <ol style="list-style-type: none">1. It's often extremely difficult for a programmer to figure out how a change in one part of the software will echo in other parts of the software <p>AND what's even worse</p> <ol style="list-style-type: none">2. Programmers sometimes change software without even trying to figure out if their changes might break something else <p>As a stage of test execution, regression testing comes after new feature testing.</p> <p>Also see <i>Bug Verification</i></p>
Release Engineer (RE)	<p>The person responsible for creating the release engineering infrastructure and for pushing code to various environments; e.g., test environments or production</p>
Required Field	<p>Web page element that should be filled with valid data (e.g., text box) or which value must be selected (e.g., value in drop-down menu) to be able to proceed to the next Web page.</p> <p>For example, in the "Email" field, valid data must have one and only one "@" character.</p> <p>Null input is always considered to be invalid for the required field.</p>
Reset Button (in HTML)	<p>Pressing this button restores the values inside a Web form back to their defaults</p> <p>For example:</p> 

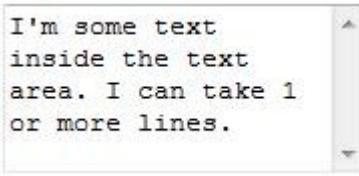
Resolution (in BTS)	<p>A stage of a bug's life</p> <p>Explanations below are given with the assumption that bug report has Type "Bug":</p> <ul style="list-style-type: none"> >Reported: A bug was filed, but the developer to fix it has not been assigned. >Assigned: Assigned developer must start bug investigation. >Fix in progress: The developer is fixing the bug. >Fixed: The bug was fixed, but the bug fix hasn't been verified yet. >Fix is verified: The bug fix has been verified. >Verification failed: The bug fix verification failed; i.e., bug is reproducible after the bug fix. >Cannot reproduce: The developer cannot reproduce the bug. >Duplicate: The bug is a duplicate of another bug. >Not a bug: The bug is not considered to represent a problem (i.e., a deviation of actual from expected). >3rd party bug: The bug is in 3rd party software. >No longer applicable: The bug doesn't have any meaning anymore.
Risk Analysis	(In the context of this Course) A black box testing technique based on evaluation of data or expectations with the purpose of setting priorities
Rollback	Action(s) to undo unwanted changes.
S	

	<p>A combination of actions and data applied to software under certain conditions.</p> <p>The purpose of a scenario is to bring test execution to the point where an actual result can be retrieved and compared with an expected result.</p> <p>Scenario</p> <p>In the example below, the verbs "Go", "Type" and "Click" point to actions. Data is presented by the word "expectations". If scenario assumes that book is in DB, then condition is: <i>DB has data about book with word "expectations" in its title</i>.</p> <ol style="list-style-type: none"> 1. Go to www.sharelane.com. 2. Type word "expectations" into the text field "Search". 3. Click the "Search" button.
Security Testing	Testing for protection against security breaches
Semi-Automated Testing	<p>Manual testing done with partial usage of the test automation, usually in form of helpers</p> <p>Also see <i>Automated Testing, Manual Testing</i></p>
Severity (in BTS)	The magnitude of a bug's impact on the software
Smoke Test	<p>(Also called "sanity test" or "confidence test") A check to determine if the software is testable</p> <p>During a smoke test, we check the main flows of the main features.</p>
Software Bug	"An error, flaw, mistake, failure, fault, or 'undocumented feature' in a computer program that prevents it from behaving as intended (e.g., producing an incorrect result)" (Wikipedia)
Software Development Life Cycle (SDLC)	<p>A way to get</p> <ul style="list-style-type: none"> - from an idea about a <i>desired software</i> - to the release of the <i>actual software</i> and its maintenance
Software Testing	A set of activities and processes primarily targeted to find AND address software bugs
Spec	The instruction on how software should work and/or look

Start-up	<p>A young company that usually has a short and eventful life</p> <p>If:</p> <ul style="list-style-type: none"> - your share in company is 0.1% or more - you eat free pepperoni pizza at least once a week - you imagine yourself to be filthy rich within 4 years <p>then it's likely that you work for a start-up. Some start-ups like Google, Netscape and YouTube did make their early employees multimillionaires.</p>
Status (in BTS)	<p>The state of a bug:</p> <ul style="list-style-type: none"> - Open - Closed - Re-open
Structured Query Language (SQL)	<p>SQL is a language to communicate with DB</p> <p>“SQL is a database computer language designed for the retrieval and management of data in relational database management systems (RDBMS), database schema creation and modification, and database object access control management” (Wikipedia).</p>
Submitted by (in BTS)	<p>The alias of the person who filed a bug</p>
Submit Button (in HTML)	<p>Button used to submit Web form to the Web server</p> <p>Example:</p> 
Summary (in BTS)	<p>A quick synopsis of the bug</p> <p>Also see <i>Description</i></p>
Syntax Bug	<p>A bug in the syntax of the software code</p> <p>Also see <i>Logical bug; UI bug</i></p>

System Testing	Functional testing of a logically complete path This term is usually applied to situations where two or more integrated components are involved. Also see <i>Component Testing; Integration Testing</i>
T	
Test Automation (TA)	<i>In general:</i> A myriad of different tools and techniques for a myriad of different purposes in software testing: code analysis, link checking, load/performance testing, code coverage, unit testing ... this list goes on and on. <i>In context of Lecture 10:</i> Test automation for regression testing. Also see <i>Helpers; Automation Scripts</i>
Test Case	<i>Conceptually:</i> an idea about verifying something – e.g., if the vacuum cleaner works, or if your taxi arrived to take you to the airport. The essential part of a test case is the expected result. <i>As a document:</i> a container with <ul style="list-style-type: none">- The scenario on one handand- The expected result on the other hand A scenario is required to bring the test case executor (the person or program) to the actual result. The expected result serves as a benchmark to compare with the actual result and conclude if they match or not. The purpose of a test case is to find a bug.

Test Case Attributes	<p>Useful parts of the test case that assist with:</p> <ul style="list-style-type: none">- Test case execution; e.g., an IDEA that clarifies what we are checking by using that test case- Test case management; e.g., SETUP AND ADDITIONAL INFO that can contain data to make test cases more maintainable <p>The most common test case attributes are:</p> <ul style="list-style-type: none">- Unique ID- Priority- IDEA- SETUP AND ADDITIONAL INFO- Revision History
Test Coverage	<p>Depending on the context, this means one of the following:</p> <ol style="list-style-type: none">1. The coverage of possible scenarios2. Test case execution coverage
Test Cycle	<p>A 2-stage process:</p> <p>Stage 1: Test preps Stage 2: Test execution</p>
Test Engineer	A professional specializing in software testing
Test Environment	The software/hardware combo where software is tested before being released to production
Test Execution	The second stage of the Test cycle
Test Plan	The master document containing information about activities regarding the testing of the certain features (or other possible subjects of testing – e.g., how the system handles load)
Test Preps	The first stage of the test cycle
Test Suite	A collection of test cases, usually dedicated to the same spec or the same feature
Test Tables	A black box testing technique that involves creating tables with inputs and/or conditions, and then combining those tables into test scenarios

Testability	Whether it makes sense to start testing For example: If a major flow (e.g., login) is not functioning, testing is blocked; therefore, we can say that the software is not testable.
Tester	See <i>Test Engineer</i>
Testing	See <i>Software Testing</i>
Text (in HTML)	The text on a Web page For example: I'm text.
Text Area (in HTML)	A multiline input field for text on a Web page For example:  A screenshot of a text area input field. Inside the field, the text "I'm some text inside the text area. I can take 1 or more lines." is displayed in a monospaced font. The text area has scroll bars on the right side.
Text Box (in HTML)	A 1-line input field for text on a Web page For example:  A screenshot of a text box input field. Inside the field, the word "expectations" is typed. To the right of the input field is a blue rectangular button with the word "Search" in white text.
The Cycle	see <i>Software Development Life Cycle</i>
Type (in BTS)	The kind of bug report: Bug Feature request
U	

UI Bug	A bug in how software presents information Also see <i>Logical Bug; Syntax bug</i>
UI Testing	The type of testing needed to find bugs in the user interface. Also see <i>UI Bug</i>
Unit Testing	"Method of testing that verifies the individual units of source code are working properly. A unit is the smallest testable part of an application. In procedural programming a unit may be an individual program, function, procedure, etc., while in object-oriented programming, the smallest unit is a method, which may belong to a base/super class, abstract class or derived/child class." (Wikipedia) Unit testing is usually performed by the programmer against his or her own freshly baked code
UNIX Timestamp	"The number of seconds elapsed since midnight Coordinated Universal Time (UTC) of January 1, 1970, not counting leap seconds" (Wikipedia)
URL	The unique address of the resource (usually a file) on the network
Usability Testing	The evaluation of the user's experience when he or she uses our software
Use Case	A description of: - How software will be (or is) used - How software must respond to certain scenarios
User Interface (UI)	The part of the software that enables a user to use the software The typical elements of a UI for Web projects are text, image, link, text field, button, etc. Software usage comes in many shapes and forms – e.g., from watching videos on YouTube to transferring money using PayPal.
V	

Valid Data	Data that should be able to assist in accomplishing some task; e.g., registration In the case of registration at ShareLane, valid data for the ZIP code is 5 digits. Any other data is invalid.
Verifier (in BTS)	The alias of the person who must verify the bug after it was fixed
Version (in BTS)	The version of the environment where the bug was found If the application version is 1.0-23/34, the version in the BTS must be 1.0
W	
Web (Word Wide Web)	"A computer network consisting of a collection of Internet sites that offer text and graphics and sound and animation resources through the hypertext transfer protocol" (definition taken from wordnet.princeton.edu)
Web Form (in HTML)	"A Web form on a Web page allows a user to enter data that is, typically, sent to a server for processing and to mimic the usage of paper forms." (Wikipedia).
Web Server	"A computer program that is responsible for accepting HTTP requests from Web clients, which are known as Web browsers, and serving them HTTP responses along with optional data contents, which usually are Web pages such as HTML documents and linked objects (images, etc.)" (Wikipedia).
Web Site	(In the context of this course) An Internet project with the UI available to users via the Web Also see <i>Web Site Architecture</i>
Web Site Architecture	The typical Web site architecture looks like this: <ul style="list-style-type: none"> - Web server (HTTP handling) - Application core (processing) - Database (storage)
White Box Testing	(Also called "glass box testing", "clear box testing", and "open box testing") The number of testing techniques that require a comprehensive

	<p>understanding of the software code</p> <p>For example:</p> <p>A programmer can perform white box testing by comparing:</p> <ul style="list-style-type: none">- The requirements from the spec- A piece of Python code from ShareLane <p>Also see <i>Black Box testing; Grey Box Testing</i></p>
White List	see <i>Dirty List - White list</i>
Workaround	Action that bypasses a problem.



Index

3

3rd party bug (bug Resolution)..... 250

4

404 - File Not Found..... 220

4Test..... 294

5

5% discount (feature)..... 168

A

A-to-E..... **204**

Absolute URL..... 220

acceptance testing....**113**, 175, 288, 296, 343, 360

achievements..... 309

action..... 156

action verbs..... 311

active search..... 316

actual result (AR)..... **10**, 34, 360

ad hoc testing..... **177**, 360

add (CVS)..... 123

alpha testing..... **165**, 360

alphanumeric characters..... 170

Also Notify (in BTS)..... **240**, 360

anchor..... 219
Apache..... **120**
Apache logs..... 165
application core..... **89**, 360
application version..... **110**, 361
architecture..... **87**
Architecture (in TA)..... 291
ASCII..... **23**, 361
Assigned (bug Resolution)..... 243
Assigned by (in BTS)..... **231**, 361
Assigned to (in BTS)..... **229**, 361
Attachment (in BTS)..... **228**, 361
attitude..... 304
automated testing..... **177**, 361
Automation scripts (in TA)..... **285**, 361

B

back end..... **45**, 361
basic Web page elements 218
beta release..... **132**
beta testing..... **132**, **166**, 362
beta testing experience..... 311
Big Four..... 314
big spenders..... **169**
black box testing..... **154**, 362
black box testing methodology..... 159, **187**, 362
bottleneck..... **164**
boundary testing..... **204**, 362

Boundary Values.....159, 169, 174, **203**
branch (in CVS).....**128**, 129
breadth.....343
broken image.....221
broken link.....220
bug.....**11**, 214, 362
Bug (bug Type).....241
bug Attributes.....**215**, 362
bug fix.....20, **244**
bug fix verification.....244
bug owner.....**229**, 363
Bug Priority Definitions.....**237**, 344
bug regression.....244
Bug Resolution Times.....**238**, 345
Bug Severity Definitions.....**234**
bug tracking.....**213**
Bug Tracking Procedure.....**251**, 344, 363
bug tracking system (BTS).....**213**, 341, 363
Bug Vault.....216
Bugzilla.....**214**
build.....**123**, 364
Build (in BTS).....**233**, 364
build id.....**125**, 364
build number.....**125**, 364
Build Schedule.....124
Build Status.....125
build_log.txt.....128
Business Requirements Document (BRD).....71

C

Cannot reproduce (bug Resolution).....245
captcha.....222
CEO.....69, 352
certificate code.....**168**
certification testing.....113
Change History (in BTS).....**241**, 364
Checkbox (in HTML).....**226**, 365
checkin (CVS).....123

checkout (CVS).....123
clear box testing.....**178**
Closed (bug Status).....241
code coverage.....279
Code Design Document.....87
code inspection.....**93**
code reuse.....292
Coding (in SDLC).....87
coding standards.....**94**, 293
coming release.....**118**
Comments (in BTS).....**233**, 365
commercial tools (CT).....293
commit.....**114**
common sense.....**15**, 146
communication.....**16**, 91
communication skills.....16
compatibility testing.....**164**
compiler.....**102**
Component (in BTS).....**232**, 365
component testing.....**168**
condition.....**24**, 156, 366
CONDITIONALLY OPEN (CVS branch status).....130
confidence test.....**111**
coverage of possible scenarios.....**160**
cross-browser testing.....**164**
cross-platform testing.....**164**
CTO.....69
CVS.....**123**

D

data.....21, 156, **366**
data (in DB).....121
data integrity.....**170**
database (DB).....87, **121**, 366
Date (in BTS).....**229**, 366
DB (in BTS).....**233**, 367
DB schema.....**121**
debugging.....**367**

degree of abstraction.....	192
deprecated.....	250
depth.....	1, 343
Description (in BTS).....	217 , 367
Dirty List - White List.....	187
DNS error - Cannot Find Server.....	220
Drop-Down Menu (in HTML).....	225 , 367
Duplicate (bug Resolution).....	248

E

E2E TA (in TA).....	286
email.....	175
entry criteria.....	263
Entry criteria (in Test plan).....	268
environment.....	87, 113, 124, 365
equivalent classes.....	201
error message.....	12, 167 , 369
examples.....	81
exit criteria.....	264
Exit criteria (in Test plan).....	268
expected result (ER).....	10 , 34, 370
exploring.....	147

F

Facebook.....	307
FAIL.....	34
feature.....	112 , 370
Feature documentation (in Test plan).....	266
feature release.....	114
Feature Request (bug Type).....	241
Fix in progress (bug Resolution).....	243
Fix is verified (bug Resolution).....	244
Fixed (bug Resolution).....	244
flask.....	246
flow.....	370
flowcharts.....	82 , 192
Flowcharts (in black box testing).....	192

Found On (in BTS).....	233 , 370
free programming languages (FL).....	293
froggy.py.....	20
front end.....	45 , 371
functional testing.....	162
functionality.....	112 , 371

G

General info (in Test plan).....	265
glass box testing.....	178
global setting.....	169
Go/No-Go criteria.....	239 , 264
Go/No-Go meeting.....	113 , 239, 296
grey box testing.....	3, 161
groups of test suites (for RT).....	274

H

Helpers (in TA).....	283 , 371
high level.....	192
history.....	32
HTTP.....	88

I

ID (in BTS).....	216 , 371
IDEA.....	35
Idea (in SDLC).....	70
Image (in HTML).....	221, 372
input.....	23
integration testing.....	168
Internet.....	372
interpreter.....	102
interview.....	319
intranet.....	40, 372
Introduction (in Test plan).....	266
invalid data.....	366
invalid input.....	21

IP address.....**126**

J

Jason Fisher.....**2, 268**

job security.....**372**

Just WOW Toilet Paper 4U, Inc.....**69**

K

keystroke.....**224**

L

length of the road.....**286**

Link (in HTML).....**219, 373**

link checking.....**177, 279**

Linked Image (in HTML).....**221, 373**

LinkedIn.....**307**

load/performance testing.....**164**

localization testing.....**163**

LOCKED (CVS branch status).....**130**

log file.....**2, 44**

logical bug.....**104**

low level.....**192**

M

mailto link.....**219**

Maintenance (in SDLC).....**134**

maintenance (in TA).....**282**

maintenance (test case).....**39**

major release.....**114**

manual testing.....**176**

Marketing Requirements Document (MRD)....**71**

mental tuning.....**304**

minor release.....**114**

misinterpretation.....**75**

misleading link.....**220**

misspelled mailto value.....**220**

mixed release.....**115**

mock-up.....**81**

N

negative testing.....**166**

new feature testing.....**112**

next release.....**118**

No longer applicable (bug Resolution).....**250**

Not a bug (bug Resolution).....**249**

O

Open (bug Status).....**241**

OPEN (CVS branch status).....**130**

open box testing.....**178**

Other things (in Test plan).....**268**

output.....**23**

P

Paint (Windows program).....**228**

PASS.....**3, 34**

passion.....**326**

passive search.....**316**

password.....**224**

Password Input Box (in HTML).....**224, 376**

patch release.....**114**

path.....**2, 4**

patterns of user behavior.....**156**

Perl.....**293**

PHP.....**293**

ping.....**126**

PjM.....**250**

positive testing.....**166**

post-release bugs.....**25**

postmortem.....**132, 363**

powerful adjectives.....**311**

pre-release bugs.....	25
presentation.....	308
prevention.....	3
Priority (in BTS).....	236, 377
procedure (of test case).....	34
processing.....	23, 89
product.....	25
Product design (in SDLC).....	71
product manager (PM).....	71, 376
Product Requirements Document (PRD).....	71
production environment (prod).....	113
Python.....	293

Q

QA engineer.....	3
QA KnowledgeBase (QA KB).....	40
quality.....	378

R

Radio Button (in HTML).....	226, 378
Re-open (bug Status).....	241
recruiter.....	3, 314
recruiting agency.....	314
regression testing.....	112
Relative URL.....	220
Release (in SDLC).....	113
Reported (bug Resolution).....	243
requirements.....	71
Reset Button (in HTML).....	227, 379
Resolution (in BTS).....	242
resume.....	308
resume template.....	311
Resumption criteria (in Test plan).....	268
revision.....	32
revision history.....	32, 37
risk analysis.....	195
Ruby.....	293

S

sanity test.....	111
scenario.....	156, 381
Schedule (in Test plan).....	266
screenshot.....	224, 228
Search.....	1, 3
security testing.....	164
self-promotion.....	316
semi-automated testing.....	176
Severity (in BTS).....	233, 381
share.....	347
SilkTest.....	279, 293
smoke test.....	259
software bug.....	2, 3, 13, 381
Software Development Life Cycle (SDLC).....	67
Software Testing Life Cycle.....	145
spec.....	12, 72
spec bug.....	13
Spec Change Procedure.....	80
spec id.....	72
spec review.....	78
standards.....	16, 150
Star.....	119
start-up.....	68, 349, 382
Status (in BTS).....	241, 382
steps (of test case).....	34
stock.....	347
stock options.....	347
Structured Query Language (SQL).....	36
sub-version.....	123
Submit Button (in HTML).....	227, 382
Submitted by (in BTS).....	229, 382
Summary (in BTS).....	216, 382
Suspension criteria (in Test plan).....	268
syntax bug.....	102
System Design Document.....	87
system testing.....	175

T

- target market.....314
test automation.....177, **279**, 383
test case.....32, 383
test case attributes.....384
test case execution.....32
test case execution coverage.....**160**
test case generation.....32
test case review.....**110**
test coverage.....160, 384
Test cycle.....384
Test Director.....214
Test Discounts.....285
Test documentation (in Test plan).....267
test engineer.....1
test environment.....**107**
test estimates.....**261**
Test execution.....259, 273
test plan.....**264**, 384
Test preps.....**184**
Test Search.....287
test suite.....**32**, 384
Testing and bug fixes (in SDLC).....111
testing environment.....124
Text (in HTML).....**219**, 385
Text Area (in HTML).....**224**, 385
Text Box (in HTML).....**222**, 385
The Big Picture Of The Cycle.....134
Things not to be tested (in Test plan).....267
Things to be tested (in Test plan).....267
Tools/Technologies (in TA).....293
trunk (in CVS).....129
Type (in BTS).....**241**, 385

U

- UI bug.....**104**
UI Testing.....**162**
unit test.....**386**
URL.....**74**, 220, 386
usability testing.....**163**
use case.....**159**
user interface (UI).....162, **386**

V

- valid data.....**366**
valid input.....**21**
valuable opinion.....**17**
venture capitalist.....69, 352
Verification failed (bug Resolution).....245
Verifier (in BTS).....**231**, 387
Version (in BTS).....**233**, 387

W

- Waterfall.....**67**
Web form.....**222**
Web Form (in HTML).....**222**, 387
Web server.....87, 120, **387**
Web site.....**387**
white box testing.....**178**
Who Does What.....**229**
Who Owns What.....**229**
WinRunner.....**293**
workaround.....**236**

Y

- Yahoo! Messenger.....69, 268

