Software Testing:
Techniques I Learned at FGCU and Microsoft

Scott Marks
Software Development Engineer in Test
Bing, Microsoft Corporation
scmarks@microsoft.com

March 31, 2011

Agenda

- Background
- Software testing
  - Advantages and challenges
  - Types
  - FGCU vs. Microsoft
- Wrap-up

Background

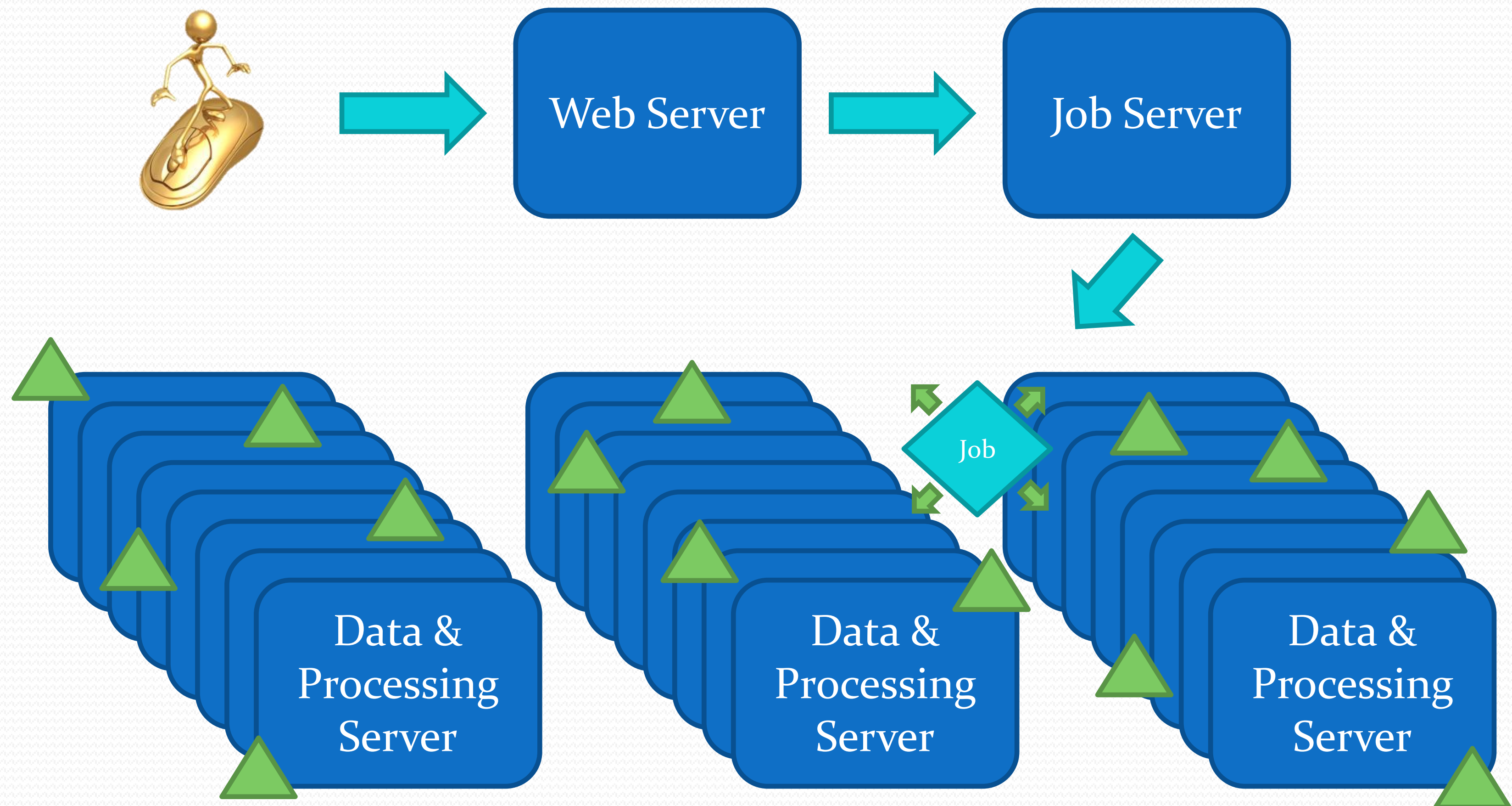
About me

- FGCU
  - Graduated in Spring 2009 with BS in Computer Science
  - Senior project
    - Multiplayer online game using HYDRA game console
  - Asteroid research
- Microsoft
  - Started in Summer 2009
  - Software Development Engineer in Test (SDET)
  - Bing's Cosmos
    - Large scale distributed storage and processing system

Cosmos distributed system

- Used internally by Bing and other Microsoft groups
- Petabyte data storage and processing system
- ~50,000 servers
- Highly reliable storage platform
- Queryable, non-relational database
- SQL-like scripting language
- Purpose:
  - Backend data processing for product features
  - Drive product innovation with anonymous click logs

Cosmos execution



Advantages and challenges

Advantages of software testing

- Minimizes risk
- Better stability and reliability
- Higher customer satisfaction
- Sooner bugs are found after creation, the easier they are to locate and resolve
- Bugs found after release can be very expensive
  - Actual cost of fix
  - Long-term cost of customer impact
  - Possible harm to users

Challenges of software testing

- Software can ***never*** be guaranteed bug-free
- Not enough resources
  - High developer-to-tester ratio
  - Near infinite combinations
  - Limited test environment
- Software requirements could be wrong or missing
- Little or no documentation
- Typical testing metrics can be grossly misleading
  - Bug counts
  - Code coverage

Problem with code coverage

```
public static int GetHighestNum(int[] nums) {
  int highest = 0;
  for (int i = 0; i < nums.Length; i++)
    if (nums[i] > highest)
      highest = nums[i];
  return highest;
}
```

= nums[0]

Throw if null
or if length == 0

```
public static bool GetHighestNumTest1() {
  return GetHighestNum(new int[] { 3, 6, 2, 4 }) == 6;
}
```

Succeeds with 100% coverage

```
public static bool GetHighestNumTest2() {
  return GetHighestNum(new int[] { -3, -6, -2, -4 }) == -2;
}
```

Failure!

Software testing

Types

Code reviews

- Informal peer review
- Find bugs without tests
- Knowledge exchange
  - Learn from others
  - Provide visibility to others
- Upholds:
  - Solid programming practices
  - Consistency with conventions

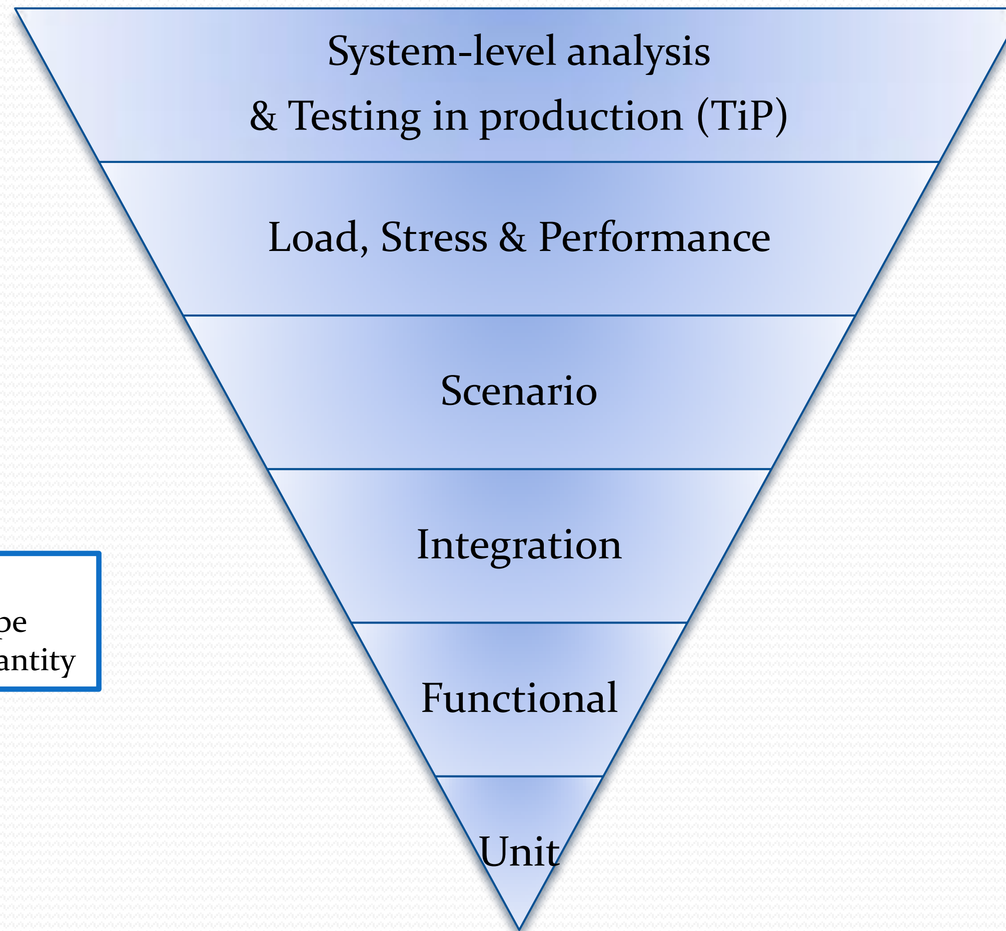
Manual vs. automated testing

	Manual	Automated
Description	Human-run tests	Software-run tests
Categories	Trial runs Bug bashes	Build verification tests Nightly tests
Measure of success	Pass/fail or subjective	Pass/fail
Cost of creation/setup	Medium	High
Cost of each run	High	Low
Cost of maintenance	Low	Usually High

Black vs. white box testing

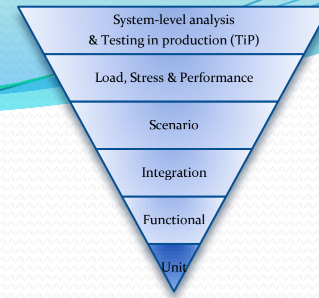
	Black box	White box
Knowledge of underlying production code	None	High
Testing bias to production code	None	High
View of software	User	Developer
Appropriate use	High-level testing: Scenario, perf, analysis, …	Low-level testing: Unit, functional

Common testing techniques



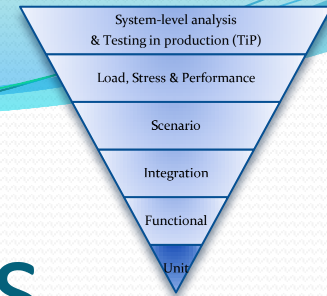
Bottom-up order:
- Increasing scope
- Decreasing quantity

Unit testing

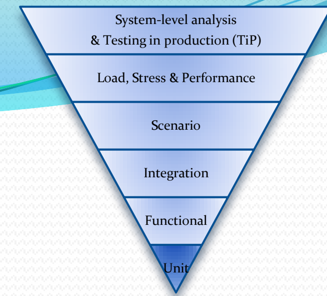


- Function/class-level testing
- Created alongside production code
  - Test-driven development
- Locate bugs quickly after creation
- Fast running
  - Entire suite should run in seconds
- Forcing function for good coding practices
  - Bad code = difficult unit tests

Some good coding practices



- Encapsulation
  - Clear separation of class responsibilities
  - Reduces software complexity
- Dependency injection
  - Functions/classes explicitly ask for things it needs
  - Avoid object creation inside business logic
    - Only use “new” in Main() and factories
- Test-driven development (TDD)
  - Iterative process
  - Create unit tests *before* creating production code
  - Write just enough production code to make the tests pass



Mock objects

- Required for proper unit testing
- Replaces dependencies during testing
- Common dependencies:
  - Network communication
  - File manipulation
  - Printing
  - Time
- Use interfaces or abstract classes
- Easier test verification

System-level analysis & Testing in production (TiP)
Load, Stress & Performance
Scenario
Integration
Functional
Unit

Example with no mocking

```
public class Printer {
  public void Print(string text) {
    /* ... */
  }
}

public void PrintText(Printer printer, string text) {
  /* ... */
  printer.Print(text);
}
```

Hard to test

Possible loss of encapsulation

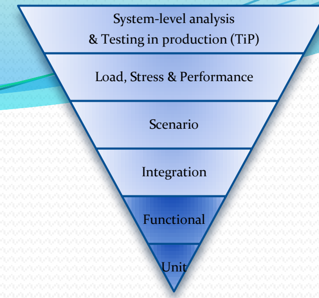
Dependency injection

Example with mocking

System-level analysis & Testing in production (TiP)
Load, Stress & Performance
Scenario
Integration
Functional
Unit

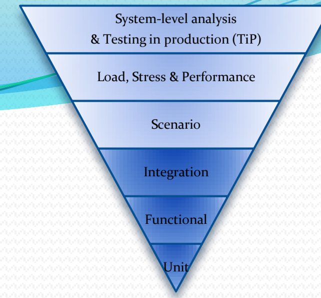
```
public interface IPrinter {
  void Print(string text);
}
public class PrinterImpl : IPrinter {
  public void Print(string text) {
    /* ... */
  }
}
public class PrinterMock : IPrinter {
  public string outputText;
  public void Print(string text) {
    outputText = text;
  }
}

public void PrintText(IPrinter printer, string text) {
  /* ... */
  printer.Print(text);
}
```

Easy to test



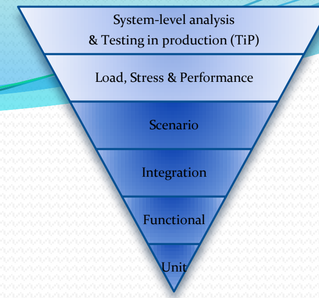
Functional testing

- Feature-level testing within a component
- Unit testing with larger scale
- Includes external dependencies
- Can simulate hard to control dependencies
- Use APIs or test hooks to verify test results
- Example:
  - Test the processing server's ability to control vertex memory usage
  - Simulate vertex memory usage and total free system memory
  - Verify vertex status via processing server's API



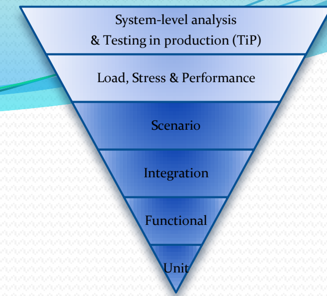
Integration testing

- Verify that 2+ binaries interact as expected
- First time components are tested outside of isolation
- Determine component dependencies early
  - Avoid silos in design, coding and testing phases
  - If neglected can cause delays or halt release
- Areas to test:
  - Exposed APIs
  - Protocols
  - Cross-component features
- Example: test that the Cosmos job manager and job vertices can communicate properly

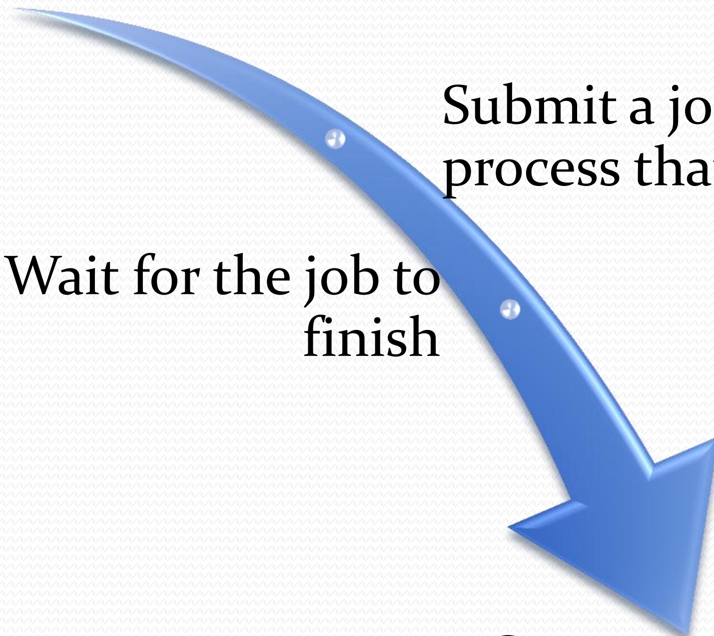


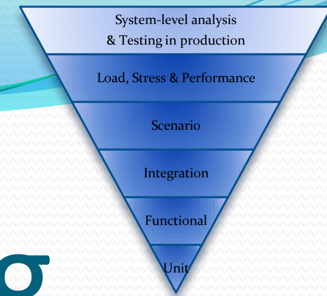
Scenario testing

- End-to-end system-level testing
- Customer story narratives
- Good, passing scenarios should be the team's primary goal
- Adds context to low-level testing
- Locate unexpected issues with sequences of features



A Cosmos user scenario

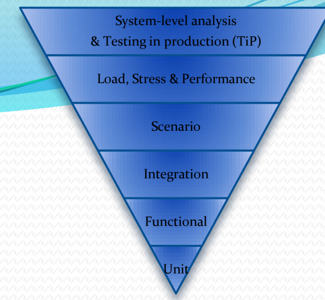




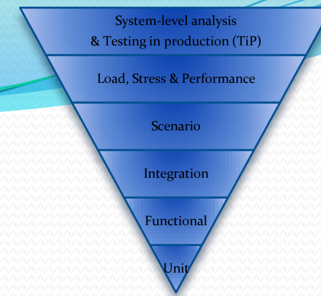
Load, stress & perf testing

	Load	Stress	Performance
Test	Run software with expected amount of work & resources	Find software's breaking point	Targeted scenarios against defined benchmarks
Confirms software…	Handles normal situations	Handles excessive load and/or limited resources	Performs at expected levels
Example	Test a multiplayer game server with a nominal number of active clients	Test a graphics-intensive game with a slow graphics card	Start a game and confirm load time meets requirements

System-level analysis

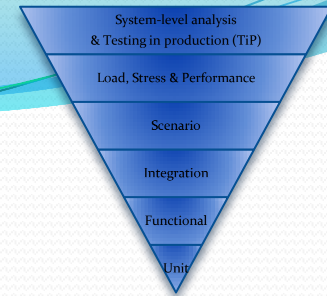


- Run the entire system to simulate production
  - Multiple scenarios at the same time
- Analyze targeted scenarios within that environment
- Metrics can determine pass/fail
- Useful when system under test is highly complex and component interactions are not well understood



Testing in production (TiP)

- Use production environments for testing
- Use real users for test input
- Locate difficult to find real-world bugs
- Drive product innovation
- Mitigate risk to users while testing



TiP practices

- Flighting
  - Small percentage of users run pre-release software
  - A/B testing
- Shadow testing
  - Users experience stable production software
  - Simultaneously run pre-release software in background with user input
- Analysis
  - Research production behaviors to find inefficiencies and bugs
  - Automated failure analysis

Other test types

- Configuration
  - Various hardware, platforms, software, etc.
- Exploratory
  - Manually explore software to locate bugs
- Usability studies
  - Get user feedback on interface and functionality
  - One-way mirrors or cameras monitor user behavior
- Beta
  - Select users get early access to pre-release software in exchange for feedback and bug notices

Software testing

FGCU vs. Microsoft

Testing at FGCU vs. Microsoft

Technique	FGCU	Microsoft
Code reviews	Occasionally	✓
Test plans	Rarely	✓
Manual tests	✓	Rarely
Automated tests	✗	✓
Unit tests	✗	✓
Scenario-driven tests	Assignment Objective	✓

Testing strategies in Cosmos

- Developer-to-tester ratio is about 2-3:1
- Test team uses different strategies

Test strategy	Developers	Testers
Feature testing	Unit tests	Functional Integration Scenario Load/stress/perf
Analysis testing	Unit tests Functional Others	Analysis Metrics TiP

Wrap-up

Summary

- Tests are necessary to understand and minimize risk
- Defect prevention with unit testing
- Testing comes in many flavors
  - Black vs. white box
  - Manual vs. automatic
  - Functional vs. analysis
- Strategy depends on the given situation
  - System complexity
  - Organization traits
  - Risk to users
- Success is difficult to quantify
  - Common metrics are misleading

Suggested materials

- *How We Test Software at Microsoft* by Alan Page, Ken Johnston, Bj Rollison
- *Software Testing* 2nd Ed. by Ron Patton
- *Your Software Has Bugs* by Seth Eliot
- *Code Complete: A Practical Handbook of Software Construction* 2nd Ed. by Steve McConnell
- Clean Code Talks by Misko Hevery
  - *Global State and Singletons*
  - *Don't Look for Things!*
  - *Inheritance, Polymorphism, & Testing*
- *Dryad* by Microsoft Research

Questions?