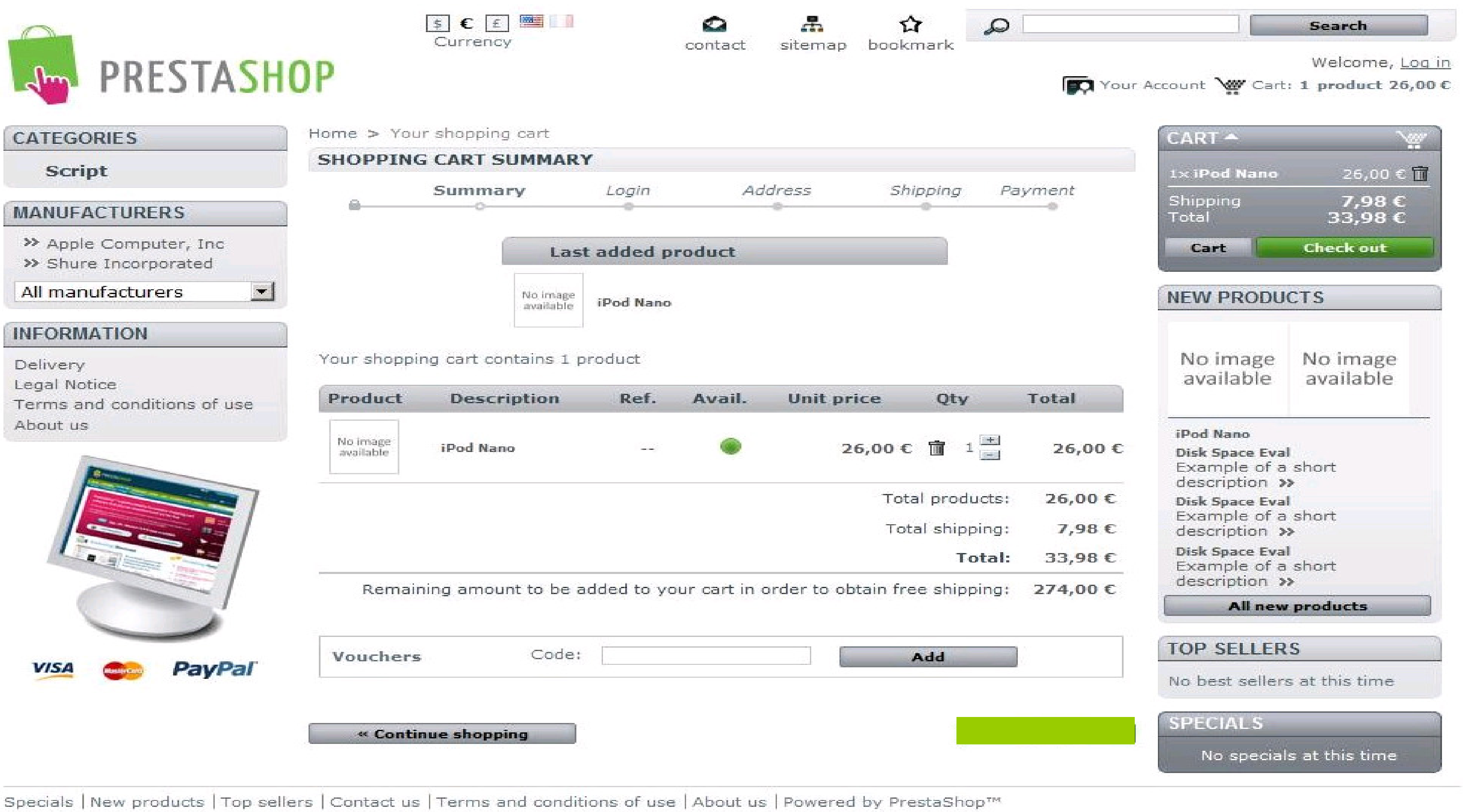
An Exploration of Errors in Web Applications in the Context of Web Application Testing

PhD Dissertation Defense

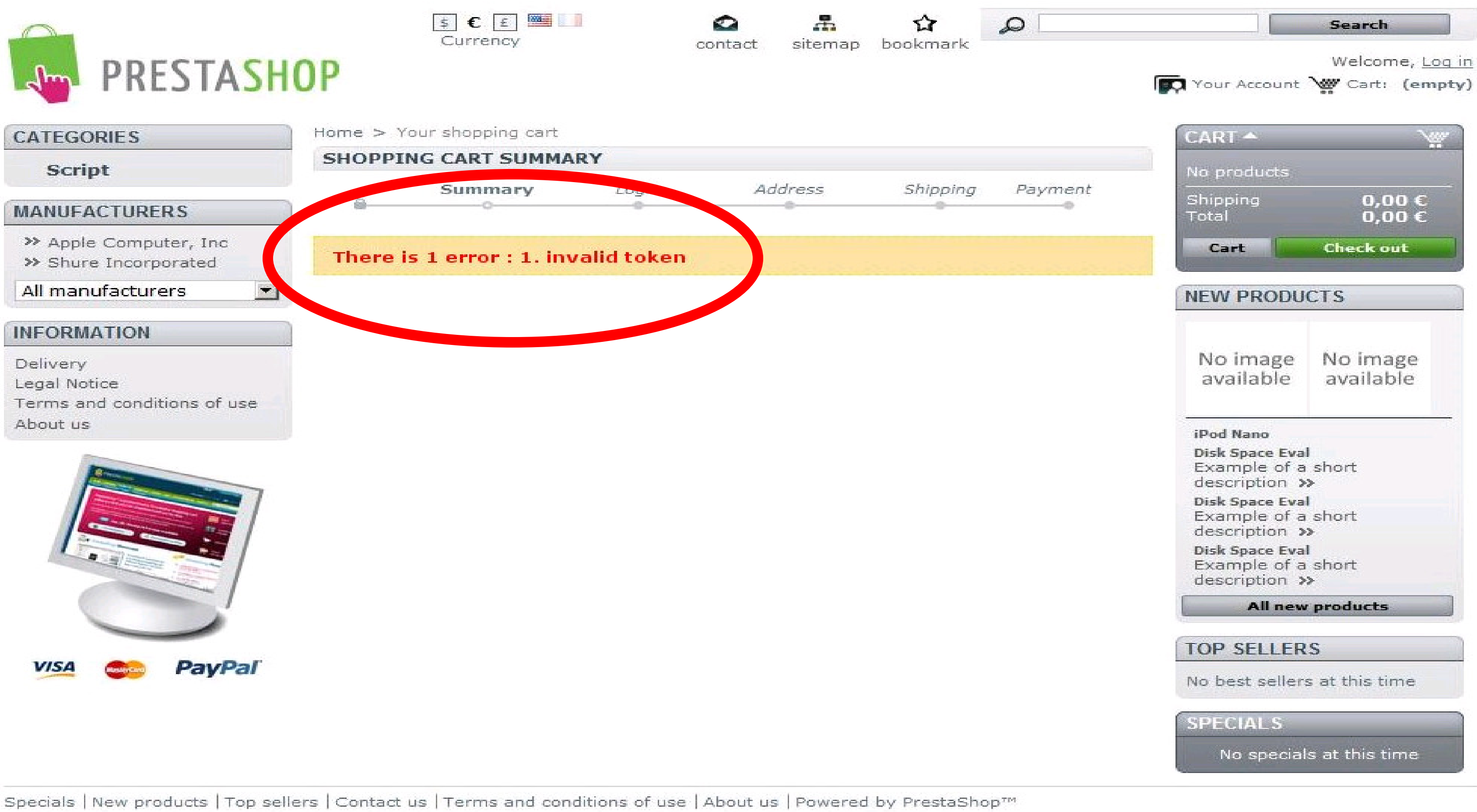
Kinga Dobolyi

April 2, 2010

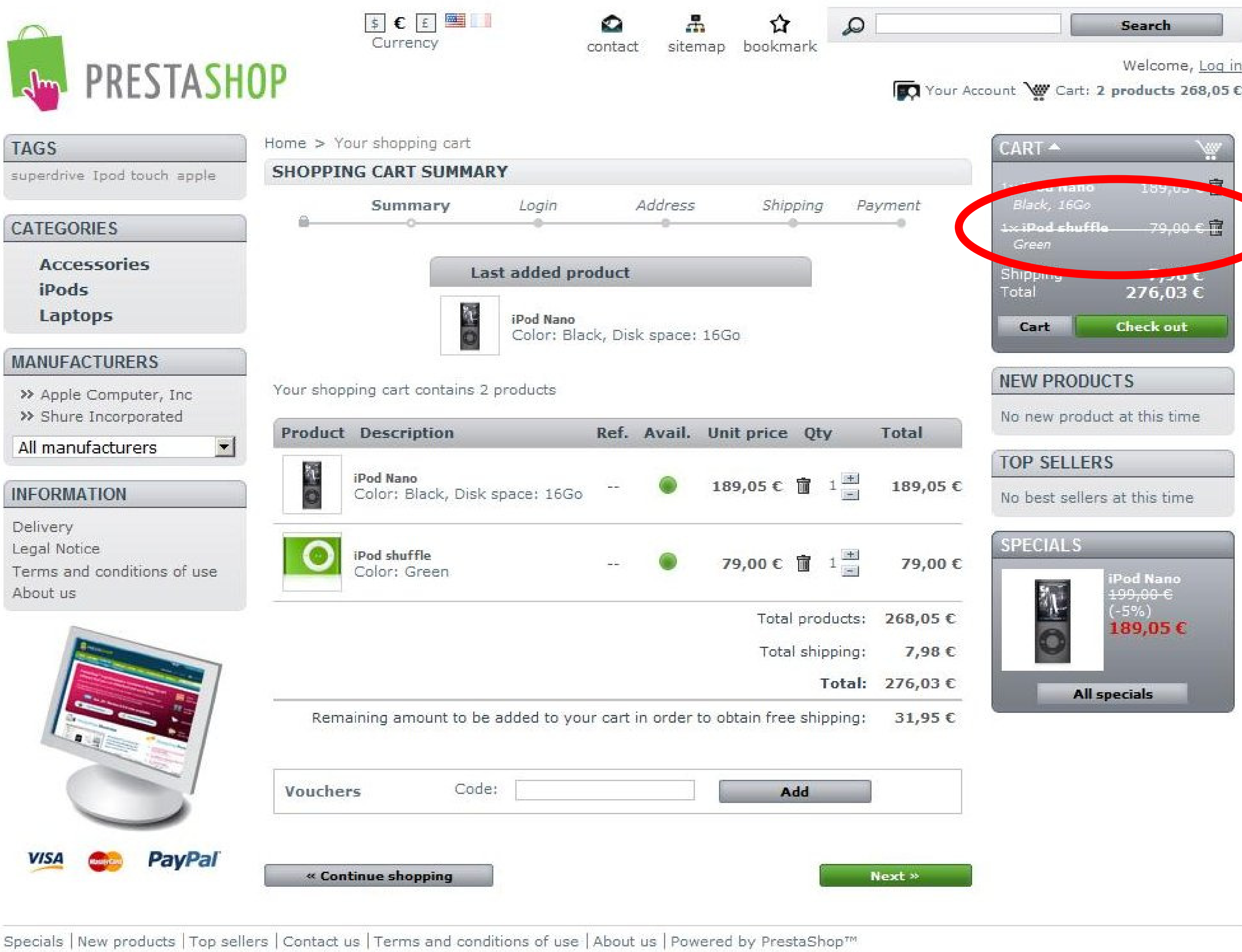
The shopping cart



The shopping cart



The shopping cart



What is going on

- **Problem**: faults in web applications cause losses of revenue, and they are hard to test
- **Approach**: explore user-visible errors in web applications to improve fault detection
- **Solution**: improve the state of the art in web testing techniques through guidelines targeted at high severity faults and automation and precision in testing

Why do we care about web application defects?

- Internet usage: 73% of people in the US in 2008
  - Browsers are dominant application
  - $204 billion in Internet retail sales annually
- Global online B2B transactions total several $*trillions* annually
- One hour of downtime at Amazon.com cost $1.5 million dollars
- 70% of major online sites exhibit user-visible failures
  - 90% of bugs reported are user-visible

Why do we care about web application defects?

- Customer loyalty is notoriously low
  - Determined by the usability of the application
  - Do not have to purchase or install software
  - Have especially high reliability, security, usability, and availability requirements

Web application Testing

- Most web applications are developed without a formal process model
- Developers often deliver the system without adequately testing it
  - Technological complications
  - Resource constraints
  - Rate of change

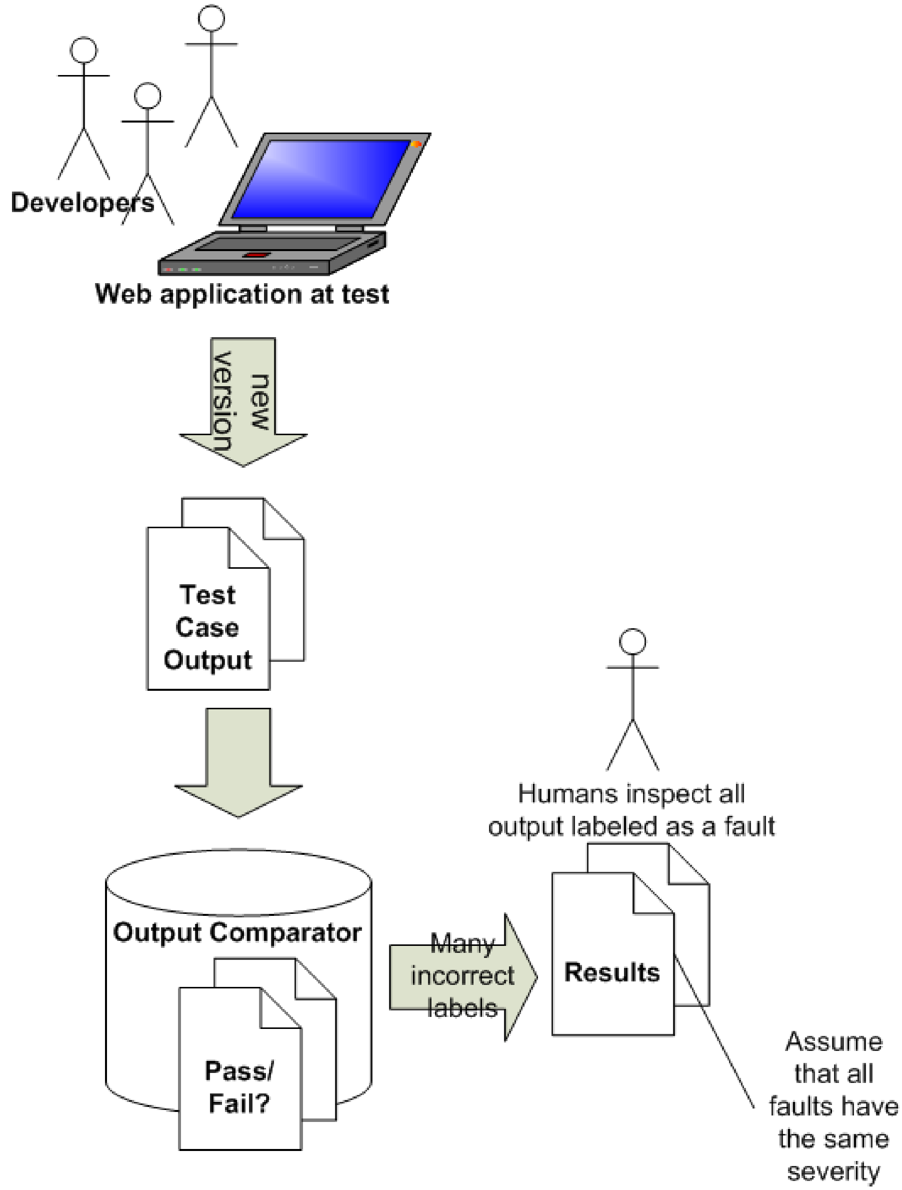
Thesis statement

- Web-based applications have special properties that can be harnessed to build tools and models that improve the current state of web application user-visible fault detection, testing, and development.
  - Tend to evolve and fail in predictable and similar ways
  - Human centric definition of acceptability

Outline

- Thesis statement
- Hypotheses H1 through H7
  1. Faults in tree-structured HTML output can be modeled
  2. Web applications fail in similar ways
  3. Not all faults are equally severe
  4. Faults can be modeled by severity
  5. Severe faults correspond to software engineering techniques
  6. Reduced test suites can preserve severe fault exposure
  7. Automated tools to detect faults rarely miss severe faults
- Summary

Current state of practice



Goals and approaches

- I propose to:
  - Model errors in web-based applications
    - Identify them more accurately
    - Automate the process of comparing expected and actual test case output
  - Make web testing more cost-effective
    - Devise a model of fault severity that will guide test case design, selection, and prioritization
    - Refute the current underlying assumption that all faults are equally severe in fault-based testing

Main Contributions

- Reduce the cost of testing web-based applications (H1, H2, and H7)
  - Provide a fully automated, highly-precise output comparator that uses special structure of web-based application output to more precisely identify errors
- Demonstrate that the assumption that injected faults have the same severity is false (H3)
  - Using a large-scale human study

Main Contributions cont'd.

- Provide human-assisted and fully automated models of fault severity (H4)
  - Reduce to cost of testing by exposing high severity faults
- Provide software engineering guidelines to decrease severe faults (H5 and H6)
  - Under the assumption of few resources during development and testing

Outline

- Thesis statement
- Hypotheses H1 through H7
  1. Faults in tree-structured HTML output can be modeled
  2. Web applications fail in similar ways
  3. Not all faults are equally severe
  4. Faults can be modeled by severity
  5. Severe faults correspond to software engineering techniques
  6. Reduced test suites can preserve severe fault exposure
  7. Automated tools to detect faults rarely miss severe faults
- Summary

Hypothesis H1

- A **highly-precise output comparator**
  - Structural and semantic features of XML\HTML output
  - **reduces the number of non-errors** flagged by naïve comparators
  - the ratio of the cost of examining a potential bug to the cost of missing an actual bug at or below a current state-of-the-art value of **0.023**.
- Model errors on a per-project basis
- Reduce false positives and false negatives
- Used during regression testing web applications

What is regression testing?

- Ensures that changes to the code do not (re-)introduce defects
- Comparing two outputs:
  – Expected output (previous, trusted version)
  – Test case output
- Comparison often accomplished with **`diff`** (a textual comparison tool)
- *Retest-all* versus reduced test suites

Comparing test output

- Oracle comparators may have difficulty with web application output

```
<P>The same table could be indented.
<TABLE border="1">
---
<p>The same table could be indented.</p>
<table border="1" summary="">
```

  - Although **`diff`** is automated, lots of false positives from **`diff`**-like tools
  - Want highly precise comparators

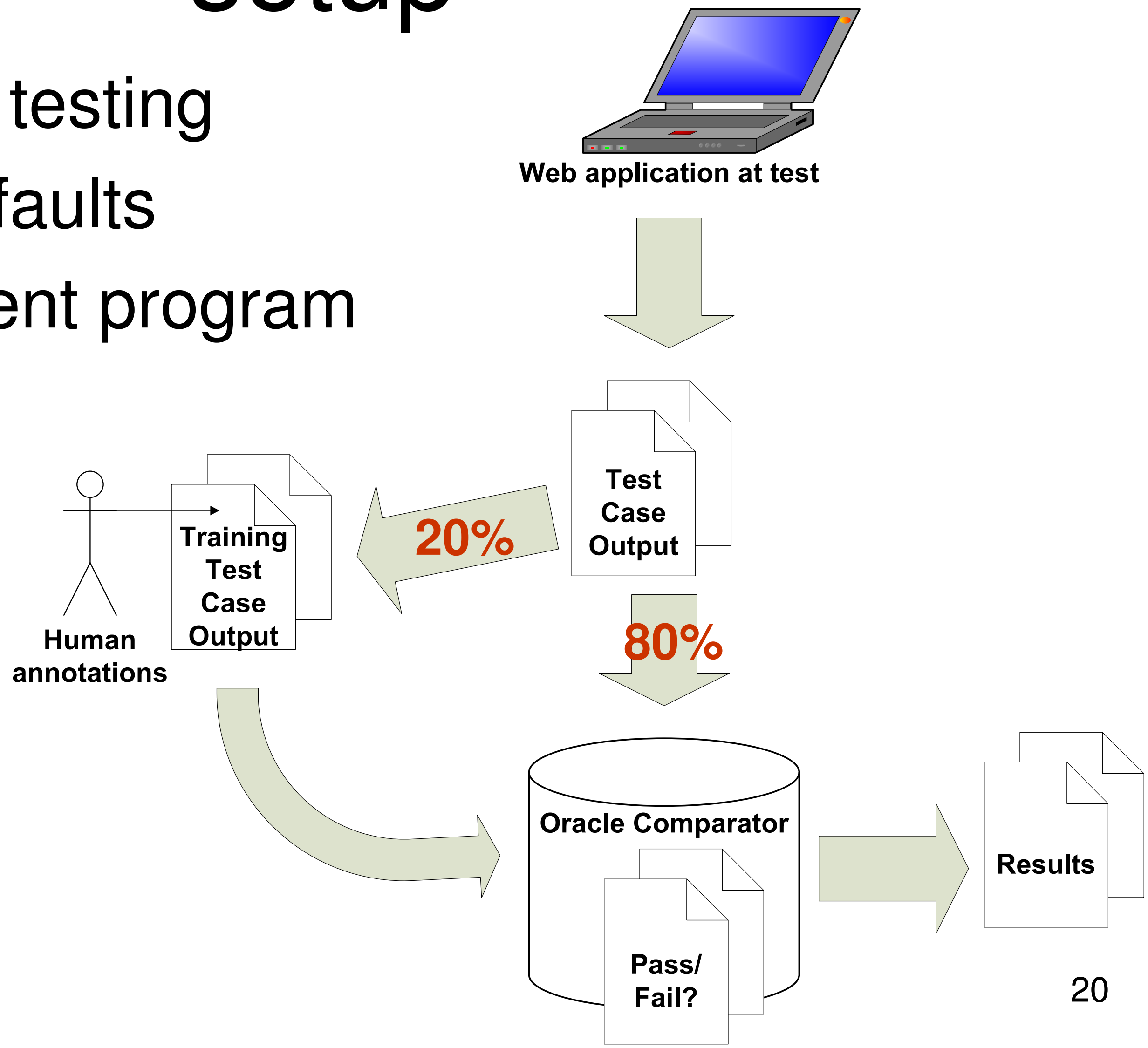
A better oracle comparator

- Model differences between test case output pairs
  - 16 surface features
    - Tree-based
    - Meaning-based
  - Use linear regression to train the comparator

Feature	Average – No Inspect	Average – Inspect
Text Ratio	0.7996	0.9636
Grouped Boolean	0.0007	0.9767
Text Only	0.9946	0.0179
Grouped Change	0.0002	0.1301
Children Order	0.0010	0.1769
Inversions	0.0010	0.0016
Depth	0.0007	0.0172
DIFF-X-delete	0.0007	0.1203
DIFF-X-insert	0.0041	0.0109
Error Keywords	0.0000	0.0096
New Text	0.6197	0.9624
New Functionality	0.0000	0.0038
Missing Attribute	0.0047	0.1580
DIFF-X-move	0.0004	0.0507
Seen Elements	0.0000	0.0014
Changed Attribute	0.5244	0.9546

Hypothesis H1: experimental setup

- Training and testing on *probable* faults across different program versions



Hypothesis H1: experimental results

- Measure effort saved:

(*TruePos* + *FalsePos*) x *LookCost* + *FalseNeg* x *MissCost* is less than |`diff`| x *LookCost*

$$\frac{LookCost}{MissCost} > \frac{-FalseNeg}{TruePos + FalsePos - |\mathtt{diff}|}$$

Goal: below 0.023

LookCost = $x

MissCost = $44x

Benchmark	Release	Test Cases	Should Inspect	True Positive SMART	True Positive diff	False Positives SMART	False Positives diff	False Negatives SMART	False Negatives diff	Ratio
HTMLTIDY	2nd	2402	12	5	12	78	781	7	0	0.0099
	3rd	2402	48	48	48	0	782	0	0	0
	4th	2402	254	109	254	1	574	145	0	0.2019
	5th	2402	48	48	48	0	775	0	0	0
	6th	2402	20	19	20	1	774	1	0	0.0013
GCC-XML	2nd	4111	662	658	662	16	2258	4	0	0.0018
	3rd	4111	544	544	544	0	2577	0	0	0
total		20232	1588	1431	1588	96	8521	157	0	0.0183

Hypothesis H1: summary

- Errors in web-based applications can be successfully modeled due to the tree-structured nature of XML/HTML output
  - Reduce false positives vs **`diff`**
  - LookCost to MissCost ratio below current state-of-the-art value of 0.023 using the oracle comparator

Outline

- Thesis statement
- Hypotheses H1 through H7
  1. Faults in tree-structured HTML output can be modeled
  2. Web applications fail in similar ways
  3. Not all faults are equally severe
  4. Faults can be modeled by severity
  5. Severe faults correspond to software engineering techniques
  6. Reduced test suites can preserve severe fault exposure
  7. Automated tools to detect faults rarely miss severe faults
- Summary

Hypothesis H2

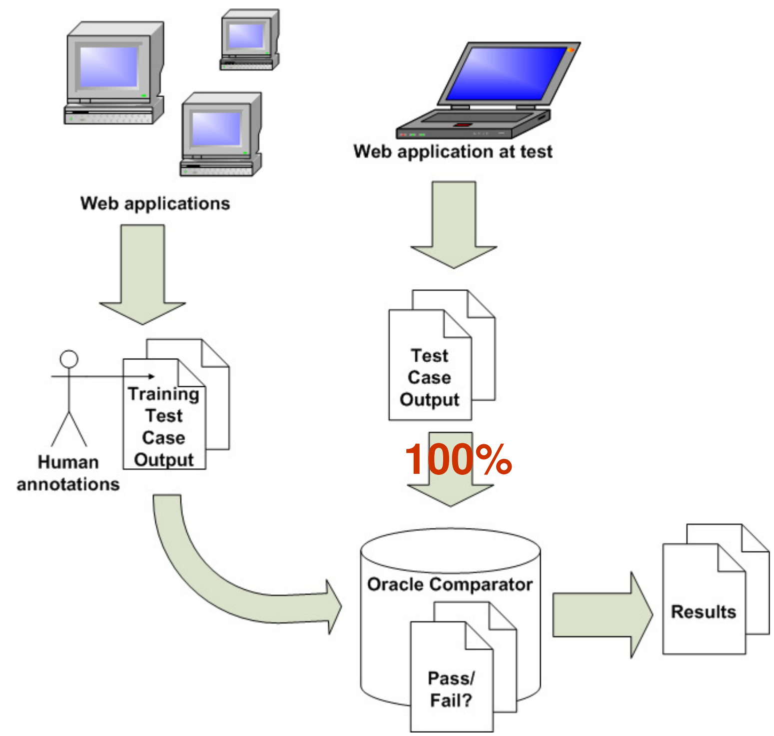
- A **highly-precise, <u>fully-automatic</u> oracle comparator**
  - based on **pre-existing information from unrelated applications**
  - **fewer false positives** than **`diff`**
  - maintaining a ratio of the cost of examining a potential bug to the cost of missing an actual bug at or below a current state-of-the-art value of **0.023**
- Train comparator on data from other, unrelated web-based applications
- Use fault injection to improve the results when necessary

What is fault injection?

- Randomly mutate one line of source code in the application, and re-run the entire test suite
  - It is assumed that any output that differs from the expected output in this case is a fault
  - Repeat until enough mutant outputs are generated

Hypothesis H2 – experimental setup

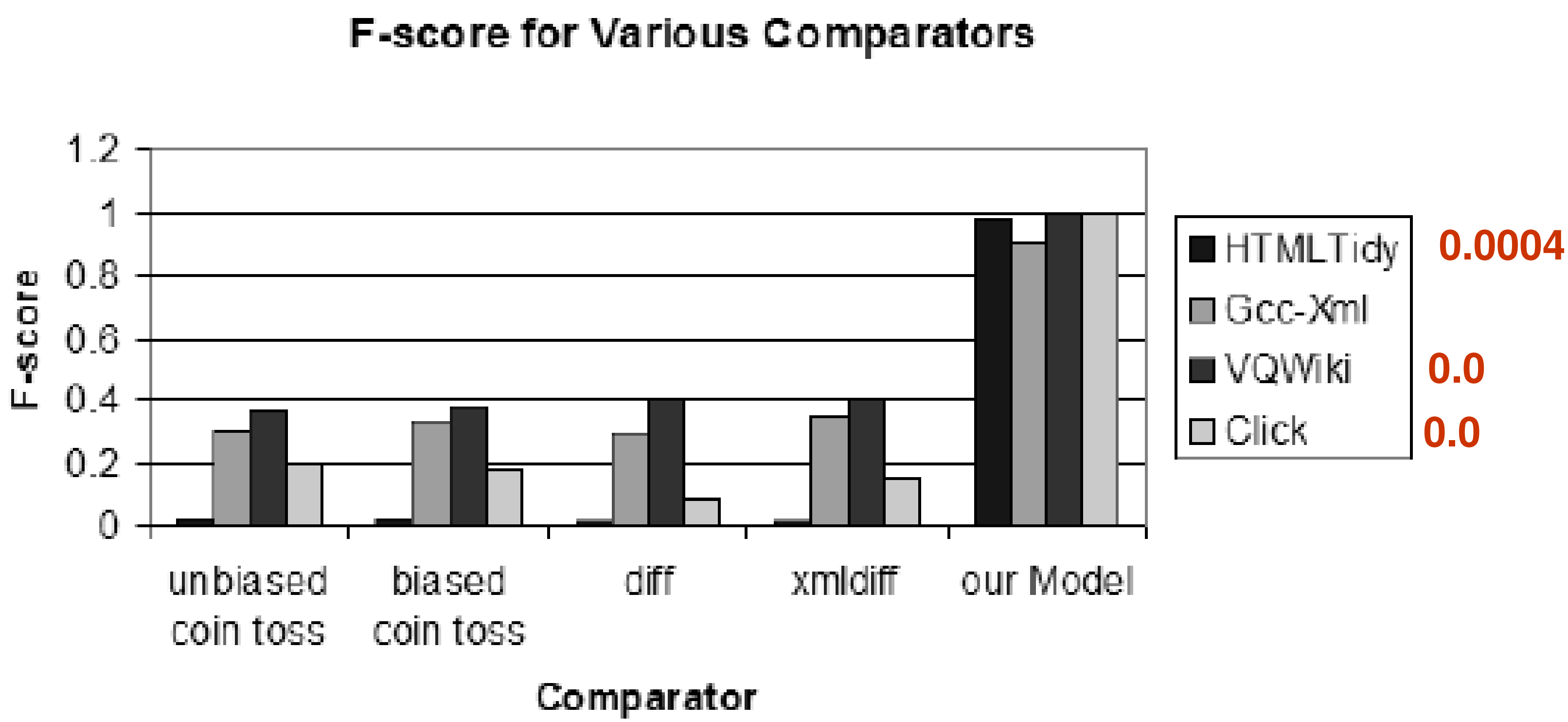
- Training Data
  - 10 web-based applications
  - Pre-annotated
- Testing Data
  - Never test and train on the same data



Benchmark	Versions	LOC	Description	Test cases	Test cases to Inspect
HTMLTIDY	Jul'05 Oct'05	38K	W3C HTML validation	2402	25
GCC-XML	Nov'05 Nov'07	20K	XML output for GCC	4111	875
VQWIKI	2.8-beta 2.8-RC1	39K	wiki web application	135	34
CLICK	1.5-RC2 1.5-RC3	11K	JEE web application	80	7
Total		108K		6728	941

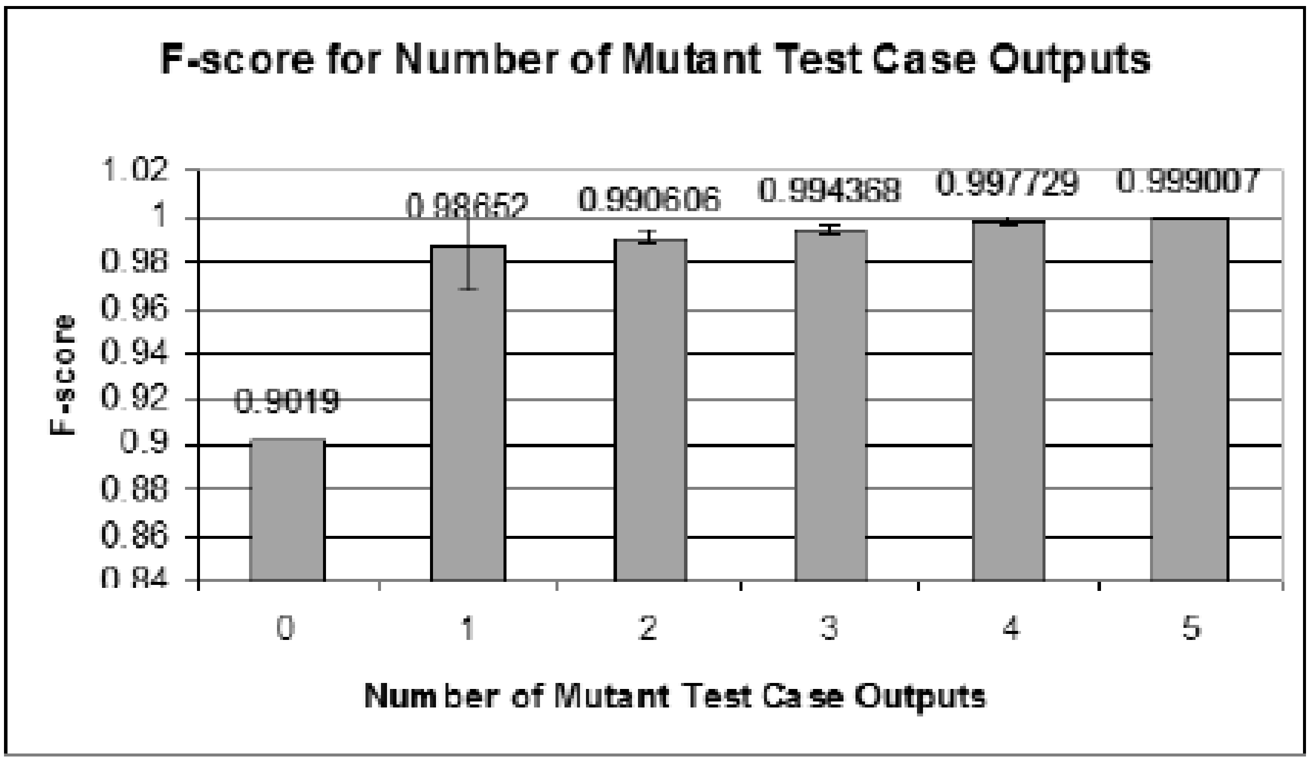
Hypothesis H2 – experimental results

- F-score is the harmonic mean of false positives and false negatives



Hypothesis H2 – experimental results

- Use fault seeding to reduce false negatives for Gcc-Xml
  - Add mutant output to training data set



Hypothesis H2: summary

- Unrelated web-based applications fail and evolve in similar ways
  - Fully automated
  - Reduce false positives vs **`diff`**
  - LookCost to MissCost ratio below current state-of-the-art value of 0.023 using the oracle comparator

Outline

- Thesis statement
- Hypotheses H1 through H7
  1. Faults in tree-structured HTML output can be modeled
  2. Web applications fail in similar ways
  3. Not all faults are equally severe
  4. Faults can be modeled by severity
  5. Severe faults correspond to software engineering techniques
  6. Reduced test suites can preserve severe fault exposure
  7. Automated tools to detect faults rarely miss severe faults
- Summary

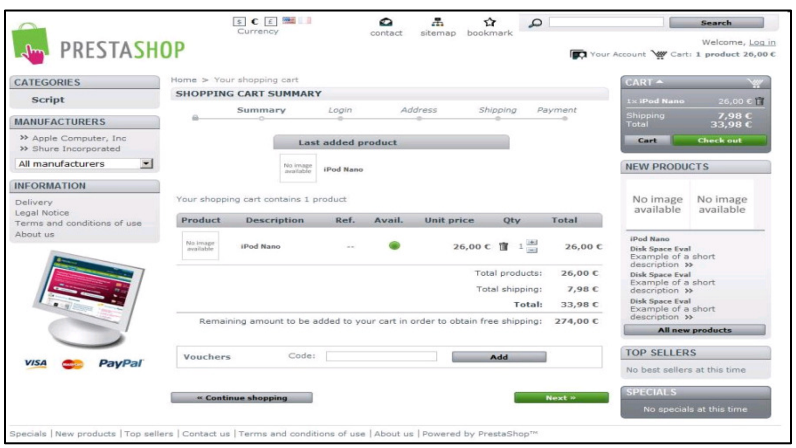
Hypothesis H3

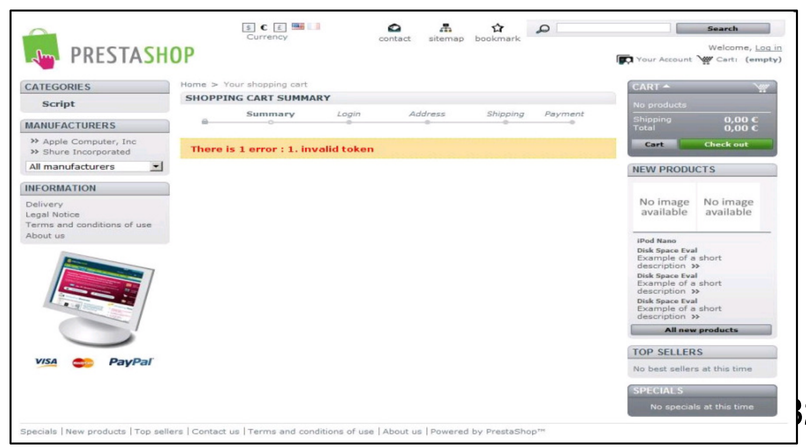
- **Faults injected** into web applications, using an automated seeding process using mutation operators, or using manual fault seeding, **vary in their underlying consumer-perceived severities**.
  - Raw fault counts may not effective in comparing competing testing approaches when considering consumer retention
  - Use a human study to measure severities
  - Consumer-perceived severity different than developer perceived severity

Hypothesis H3: Human study setup

- Each fault presented as a scenario triple:
  - *current* screenshot
  - scenario description
  - *next* screenshot

Description	Severity Rating
I did not notice any fault	0
I noticed a fault, but I would return to this website again	1
I noticed a fault, but I would probably return to this website again	2
I noticed a fault, and I would not return to this website again	3
I noticed a fault, and I would file a complaint	4





Hypothesis H3: Human study results

- Large scale human study
  - 400 real-world faults, 400 injected faults, 100 non-faults from 17 real-world web applications
  - 386 subjects
  - Over 12,000 votes (at least 12 per fault)

Fault Type	Low ≤ 1	Med ≤ 2	Med-High < 2.5	Severe ≥ 2.5
Real-world	23%	30%	19%	28%
Automatic-injected	25%	25%	27%	23%
Manual-injected	23%	28%	27%	22%
Non-fault	92%	7%	0%	1%

Hypothesis H3: summary

- Not all failures in web applications have the same consumer-perceived severity
  - Both injected and manual faults vary in their severity levels

Outline

- Thesis statement
- Hypotheses H1 through H7
  1. Faults in tree-structured HTML output can be modeled
  2. Web applications fail in similar ways
  3. Not all faults are equally severe
  4. Faults can be modeled by severity
  5. Severe faults correspond to software engineering techniques
  6. Reduced test suites can preserve severe fault exposure
  7. Automated tools to detect faults rarely miss severe faults
- Summary

Hypothesis H4

- An **automated model of consumer-perceived fault severity** can be constructed that agrees with human severity judgments **at least as often as humans agree with each other**, evaluated using the Spearman's Ranking Correlation Coefficient (SRCC)
  - Can be used to prioritize faults

Hypothesis H4: background

- Consumer perceived fault severity is poorly understood
  - Do not rely on individual human judgments of fault severity, as these can be inaccurate
- Want to make testing more efficient by targeting *consumer* perceived fault severity
  - Agree with humans at least as often as they agree with each other

Hypothesis H4: Modeling fault severity

- Build a model of consumer-perceived fault severity
  - Using 17 boolean surface features of faults
    - Stack traces, missing images, cosmetic errors, SQL code, authentication, etc.
  - A human-assisted model that uses human annotations of rendered browser output
  - A fully automated model that examines pairs of HTML output

Hypothesis H4 – experimental results

- Both models are better than humans on average at correctly predicting fault severity

Model	Accuracy	Severe Missed	Non-Severe Correct	
Automated Model	83%	0/30	39/70	0.78
Annotation-based Model	84%	1/30	61/70	0.84
Individual Human (avg)	59%	8/30	53/70	0.70
Always Average Rating	58%	30/30	70/70	0.51
Always Median Rating	59%	30/30	70/70	0.51
C4.5 Decision Tree	85%	5/30	65/70	0.76

Hypothesis H4: summary

- Faults in web applications can be modeled according to their consumer-perceived severities
  - Agrees with average human judgments of severity more often than humans agree with each other
  - Fully automated

Outline

- Thesis statement
- Hypotheses H1 through H7
  1. Faults in tree-structured HTML output can be modeled
  2. Web applications fail in similar ways
  3. Not all faults are equally severe
  4. Faults can be modeled by severity
  5. Severe faults correspond to software engineering techniques
  6. Reduced test suites can preserve severe fault exposure
  7. Automated tools to detect faults rarely miss severe faults
- Summary

Hypothesis H5

- There exists a **statistically significant correlation** (SRCC > 0.60) between **severe faults** in web applications and various **software engineering aspects** of web application development
  - Can be used when there are few to no resources for testing

Hypothesis H5: experimental setup

- Analyze the data from the large-scale human study to look for correlation between high severity faults and
  - The type of web application
  - The visual presentation of the defect
  - The source of the defect in the code

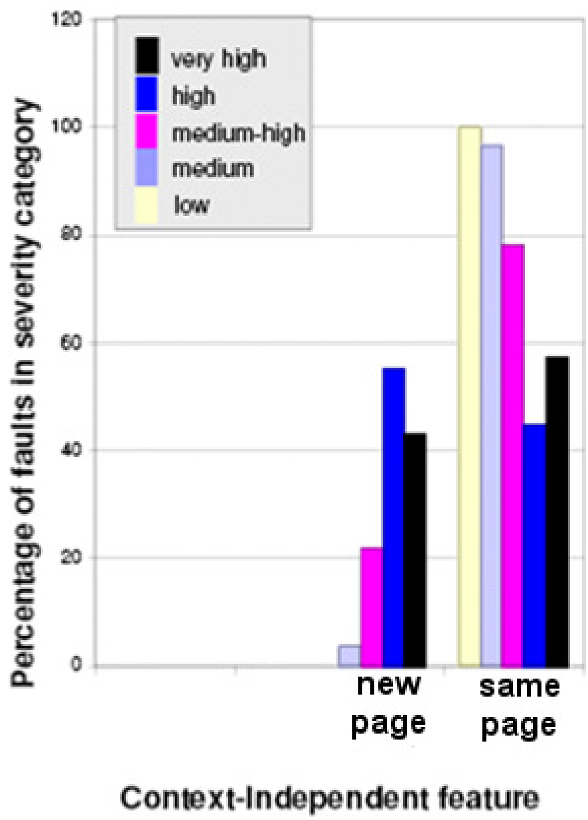
Hypothesis H5: application types

- As a baseline, little to no correlation between the type of application or the programming languages used and severe faults

Feature	SRCC with severe faults
is written in PHP	0.16
is written in ASP.net	0.32
is a Gallery	0.38
is a Wiki	0.35
is a Forum	0.34
is a Content Mgmt. System	0.30
is E-commerce	0.22

Hypothesis H5: fault visualization

- Keep the appearance of the page the same
- Opt for popups over server generated error messages or stack traces



Hypothesis H5: fault causes

- Many classes of faults are associated with high severity
  - Even a naïve test suite can detect many such faults

Fault Cause	SRCC with severe faults
Configuration	0.68
SQL	0.69
Permissions	0.68
Server	0.68
NULL	0.69
Component	0.62
Database	0.66
Upgrade	0.63
Other error in source code	0.18

Hypothesis H5: summary

- Severe faults correspond to specific software engineering aspects during web application development
  - Statistically significant correlation

Outline

- Thesis statement
- Hypotheses H1 through H7
  1. Faults in tree-structured HTML output can be modeled
  2. Web applications fail in similar ways
  3. Not all faults are equally severe
  4. Faults can be modeled by severity
  5. Severe faults correspond to software engineering techniques
  6. Reduced test suites can preserve severe fault exposure
  7. Automated tools to detect faults rarely miss severe faults
- Summary

Hypothesis H7

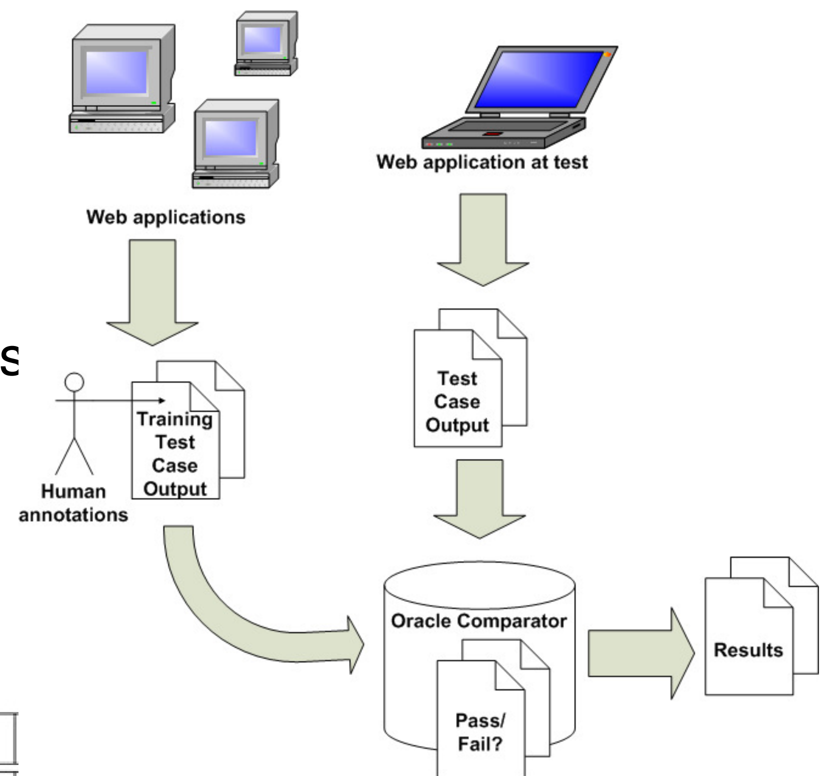
- **At most 1% of the false negatives** produced by the highly-precise, fully-automated oracle comparator correspond to **severe faults**
  - Would we want to use this tool in the real world?

Hypothesis H7: approach

- Combine automated comparator with fault severity model
- Evaluate automated approach on 3 real-world, *popular* PHP benchmarks
  - Known (seeded faults)
  - Heavy use of non-deterministic output
  - Measure fault severity of missed faults

Hypothesis H7: experimental setup

- Train the oracle comparator as before
  - On data from unrelated web applications
  - Use automatically seeded faults as additional training data
  - Use a clean run of the test suite as additional training data



Benchmark	Versions	LOC	Description
PRESTASHOP	v1.1.0.55	155K	e-commerce (shopping cart)
OPENREALTY	v2.5.6	185K	real estate listing management
VANILLA	v1.1.5a	35K	web forum
Total		375K	

Training Faults	Testing Faults	Test Suite
5401	955	164

Hypothesis H7: experimental results

- Correctly identified 70% of non-faults
  - `diff` would get 0%
- Correctly identified 99% of severe faults
- Requires no manual annotation or training

Benchmark	Goal Severe Faults	Goal Medium Faults	Goal Low Faults	Goal V Low Faults	Miss Severe Faults	Miss Medium Faults	Miss Low Faults	Miss V Low Faults	Weighted % Found Faults	% Found Non-Faults (savings)
PRESTASHOP	302	45	57	27	3	32	17	26	98%	47%
OPENREALTY	44	1	1	16	4	1	0	10	85%	100%
VANILLA	186	183	10	83	0	5	8	0	89%	97%

Hypothesis H7: summary

- Automated tools that detect failures rarely miss severe faults
  - The highly-precise, fully-automated oracle comparator missed only 1% of severe faults on average

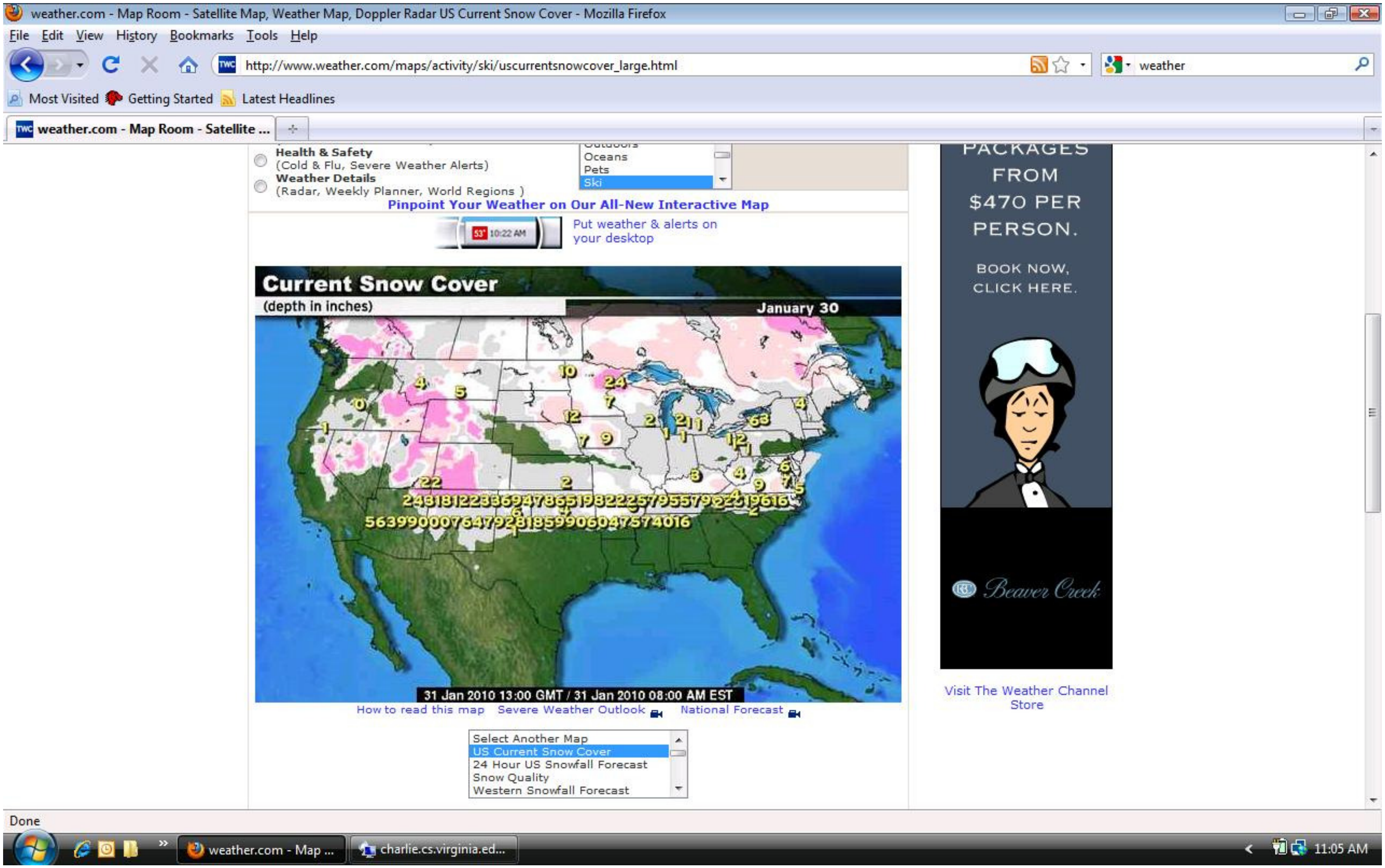
Summary

- Web-based applications have special properties that can be harnessed to build tools and models that improve the current state of web application fault detection, testing, and development
  - First to provide a fully automated oracle comparator
  - First to provide a fully automated model of fault severity
  - Software engineering guidelines to reduce fault severity
- Strong results on real-world web applications

Conclusion

- **Problem**: faults in web applications cause losses of revenue, and they are hard to test
- **Approach**: explore user-visible errors in web applications to improve fault detection
- **Solution**: improve the state of the art in web testing techniques through guidelines targeted at high severity faults and automation and precision in testing

Questions?



weather.com - Map Room - Satellite Map, Weather Map, Doppler Radar US Current Snow Cover - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://www.weather.com/maps/activity/ski/uscurrentsnowcover_large.html
weather
Most Visited Getting Started Latest Headlines
weather.com - Map Room - Satellite ...
Health & Safety
(Cold & Flu, Severe Weather Alerts)
Weather Details
(Radar, Weekly Planner, World Regions)
Oceans
Pets
Ski
Pinpoint Your Weather on Our All-New Interactive Map
Put weather & alerts on your desktop
Current Snow Cover
(depth in inches)
January 30
31 Jan 2010 13:00 GMT / 31 Jan 2010 08:00 AM EST
How to read this map
Severe Weather Outlook
National Forecast
Select Another Map
US Current Snow Cover
24 Hour US Snowfall Forecast
Snow Quality
Western Snowfall Forecast
PACKAGES FROM $470 PER PERSON.
BOOK NOW, CLICK HERE.
Beaver Creek
Visit The Weather Channel Store
Done
weather.com - Map ...
charlie.cs.virginia.ed...
11:05 AM

Further reading

- H1: highly-precise comparator:
  - Elizabeth Soechting, Kinga Dobolyi and Westley Weimer. **Syntactic Regression Testing for Tree-Structured Output**. *Web Systems Evolution, September 2009. (*invited to special section in the *International Journal on Software Tools for Technology Transfer)*
- H2: automated comparator:
  - Kinga Dobolyi and Westley Weimer. **Harnessing Web-based Application Similarities to Aid in Regression Testing**. *International Symposium on Software Reliability Engineering, November 2009.*
- H3 and H4: fault severity models:
  - Kinga Dobolyi and Westley Weimer. **Modeling Consumer-Perceived Web Application Fault Severities for Testing**. Submitted, *International Symposium on Software Testing and Analysis*
- H5 and H6: guidelines for reducing fault severity:
  - Kinga Dobolyi and Westley Weimer. **Addressing High Severity Faults in Web Application Testing**. *The IASTED International Conference on Software Engineering, February 2010.*
- H7: highly-precise automated comparator on challenging webapps:
  - Kinga Dobolyi, Elizabeth Soechting, and Westley Weimer. **Harnessing Web-based Application Similarities to Aid in Automated Regression Testing.** Invited paper; submitted, *International Journal on Software Tools for Technology Transfer*

Web Failure

- “the inability to obtain and deliver information, such as documents or computational results, requested by web users.” – Ma and Tian

Manual Fault Seeding for web applications

- Five categories (as in Sprenkle et al.):
  - Database
  - Logic
  - Form
  - Appearance
  - Link

Why is web testing hard?

- Interfaces are difficult to identify
  - Depends on user inputs and data not accessible through web forms
  - Difficult to interact with application so that all forms are exercised
  - Interfaces cannot be extracted by a simple local analysis or spider-like tools
- Control flow depends on individual usage patterns
  - Subsequent actions depend on previous user input
- Heterogenous components make modeling and static analysis difficult
  - Static analysis hard for dynamic langugaes such as PHP which enables creation of code and overriding methods on the fly
  - Def-use chains need to be extended across client/server boundaries

How are GUIs tested?

- None (most common)
- Bypassing GUI that requires major changes to the architecture
- Manual tools that provide little automation
- Capture-replay

GUI testing and web applications

- GUI and web application similarities
  - Are event-based systems operating on state
  - Difficult to create oracles for verbose output
- GUI and web application differences [Memon]
  - GUIs produce deterministic graphical output
  - Web applications have synchronization/timing constraints among objects
  - Web applications are tightly coupled with back end code (i.e. their content is dynamically created using a database)

Hypothesis H6

- There exist **test suite reduction** strategies that **expose at least 90% of the severe faults** found via corresponding retest-all approaches for web applications
  - Identify testing techniques to maximize return on investment by targeting high-severity faults

Hypothesis H6 – experimental setup

- Measure fault severity preservation of test suite reduction approaches
  - 90 manually seeded faults in 3 PHP benchmarks
  - 3x50 user sessions collected from volunteers
  - Implement 3 testing strategies:
    - *Retest-all* (baseline)
    - *HGS*: Harrold-Gupta-Soffa
    - *Concept*: Sprenkle et al.
  - Define testing requirements as URLs visited

Hypothesis H6 – experimental results

- *HGS* and *Concept* continue to be effective when considering fault severity

Method/ Benchmark	Test Cases	Low	Med	Med -High	High	Total
retest-all Prestashop	50	0	3	24	3	30
HGS Prestashop	8	0	3	24	3	30
Concept Prestashop	27	0	3	24	3	30
retest-all Openrealty	50	1	3	1	23	28
HGS Openrealty	15	1	3	1	20	25
Concept Openrealty	40	1	3	1	23	28
retest-all Vanilla	50	5	22	2	1	30
HGS Vanilla	4	5	22	2	1	30
Concept Vanilla	9	5	22	2	1	30

Hypothesis H6: summary

- Test suites can be reduced in size while preserving severe fault exposure
  - Reduced test suites exposed at least 90% of the severe faults

HGS test suite reduction

- Test cases are associated with the requirement they meet
- The number of test cases that cover a requirement is the requirement's cardinality
- Add a test case to the reduced set, marking the covered requirements
  - Select next test case to add that covers the most unmarked requirements (i.e. the lowest requirement cardinality)

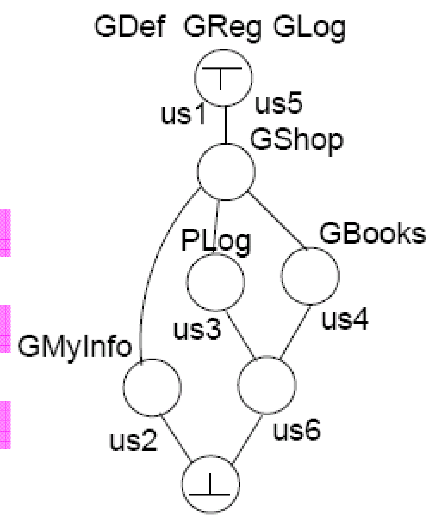
us	GDef	GReg	GLog	PLog	GShop	GBooks	GMyInfo
1	●	●	●				
2	●	●	●		●		●
3	●	●	●	●	●		
4	●	●	●		●	●	
5	●	●	●				
6	●	●	●	●	●	●	

Concept test suite reduction

- Test cases are associated with the requirement they meet (a URL)
- Build a concept lattice
  - Edges of lattice are partial ordering of concept nodes

us	GDef	GReg	GLog	PLog	GShop	GBooks	GMyInfo
1	●	●	●				
2	●	●	●		●		●
3	●	●	●	●	●		
4	●	●	●		●	●	
5	●	●	●				
6	●	●	●	●	●	●	

(a) Original Suite of User Sessions



(b) Concept Lattice

Web-based application

- A web-based application is different from a web application in that web-based applications may output XML code that does not necessarily end up rendered by a browser (i.e. such as web services that communicate through XML)

Training Benchmarks

- For automated oracle comparators

Benchmark	Versions	LOC	Description	Test cases	Test cases to Inspect
HTMLTIDY	Jul'05 Oct'05	38K	W3C HTML validation	2402	25
LIBXML2	v2.3.5 v2.3.10	84K	XML parser	441	0
GCC-XML	Nov'05 Nov'07	20K	XML output for GCC	4111	875
CODE2WEB	v1.0 v1.1	23K	pretty printer	3	3
DOCBOOK	v1.72 v1.74	182K	document creation	7	5
FREEMARKER	v2.3.11 v2.3.13	69K	template engine	42	1
JSPPP	v0.5a v0.5.1a	10K	pretty printer	25	0
TEXT2HTML	v2.23 v2.51	6K	text converter	23	6
TXT2TAGS	v2.3 v2.4	26K	text converter	94	4
UMT	v0.8 v0.98	15K	UML transformations	6	0
Total		473K		7154	919

SMART global results

Comparator	F_1-score	Precision	Recall
SMART	0.9931	0.9972	0.9890
SMART w/ cross-validation	0.9935	0.9951	0.9920
`diff`	0.3004	0.1767	1.0000
`xmldiff`	0.2406	0.1368	1.0000
fair coin toss	0.2045	0.1286	0.4984
biased coin toss	0.2268	0.1300	0.8868

Feature	Coefficient	F	p
Text Only	- 0.288	168970	< 0.001
DIFF-X-move	+ 0.002	150840	< 0.001
DIFF-X-delete	+ 0.029	46062	< 0.001
Grouped Boolean	+ 0.714	7804	< 0.001
DIFF-X-insert	+ 0.029	4761	< 0.001
Grouped Change	- 0.012	465	< 0.001
Children Order	- 0.002	317	< 0.001
Inversions	+ 0.001	246	0.020
Missing Attribute	- 0.048	121	< 0.001
Error Keywords	+ 0.174	115	< 0.001
Depth	- 0.000	21	< 0.001
Text Ratios	- 0.007	18	< 0.001
Input Elements	- 0.019	5	0.03

SMART per-project results

Benchmark	Comparator	F_1	Precision	Recall
HTMLTIDY	SMART	1.000	1.000	1.000
	diff	0.048	0.025	1.000
	xmldiff	0.021	0.010	1.000
GCC-XML	SMART	0.999	1.000	0.999
	diff	0.352	0.213	1.000
	xmldiff	0.352	0.213	1.000
All ten (global)	SMART	0.993	0.997	0.989
	diff	0.300	0.177	1.000
	xmldiff	0.241	0.138	1.000

Web application benchmarks

Name	Language	Description	Faults
Prestashop*	PHP	e-commerce	30
Dokuwiki*	PHP	wiki	30
Dokeos	PHP	e-learning	22
Click*	Java	JEE webapp framework	3
VQwiki*	Java	wiki	6
OpenRealty*	PHP	real estate listing management	30
OpenGoo	PHP	web office	30
Zomplog	PHP	blog	30
Aef	PHP	forum	30
Bitweaver	PHP	content mgmt framework	30
ASPgallery	ASP.NET	gallery	30
YetAnother Forum	ASP.NET	forum	30
ScrewTurn	ASP.NET	wiki	30
Mojo	ASP.NET	content mgmt system	30
Zen Cart	PHP	e-commerce	30
Gallery	PHP	gallery	30
other	-	-	9

2-way ANOVA for human study

Factor	F value	p value
Rating	592	< 0.001
Human Rater	5	< 0.001

Group 1	Group 2	D value	p value
Real-world	Automatically-injected	0.0691	0.524
Real-world	Manually-injected	0.0969	0.170
Automatically-injected	Manually-injected	0.0616	0.839
Real-world	Non-fault	0.7211	< 0.001
Automatically-injected	Non-fault	0.7154	< 0.001
Manually-injected	Non-fault	0.7083	< 0.001

Figure 7.5: Kolmogorov-Smirnov test results for the dataset.

Fault type comparison

Benchmark	≤ 1	≤ 2	< 2.5	≥ 2.5
PRESTASHOP real-world	37%	27%	17%	20%
PRESTASHOP automatically-injected	24%	45%	24%	6%
PRESTASHOP manually-injected	30%	33%	21%	15%
OPENREALTY real-world	41%	21%	31%	7%
OPENREALTY automatically-injected	21%	24%	18%	36%
OPENREALTY manually-injected	45%	24%	27%	3%
DOKUWIKI real-world	22%	22%	11%	44%
DOKUWIKI automatically-injected	36%	27%	24%	12%
DOKUWIKI manually-injected	12%	27%	21%	39%
VQWIKI real-world	0%	33%	33%	33%
VQWIKI automatically-injected	21%	24%	31%	24%
VQWIKI manually-injected	36%	32%	14%	18%

Developer survey results

LOC	Low	Med	Med-High	Severe
200K	57%	12%	8%	23%
2,000K*	38%	40%	15%	7%
n/a	90%	5%	3%	2%
20K	90%	10%	0%	0%
n/a	95%	3%	1%	1%
1K	75%	12%	12%	1%
n/a	85%	10%	0%	5%
$>$ 100K	78%	14%	2%	6%
n/a	86%	10%	0%	3%
1K	90%	10%	0%	0%
Average	42%	36%	14%	8%

Boolean fault surface features

		% of Faults
Arithmetic Calculation Error	Generally for shopping-cart based applications, any error in calculating the amount paid, shipping, taxes, discount applied, quantities ordered, etc.	3
Blank Page	An empty page containing no information or text.	2
404 Error	An error experienced when the URL is not found; the words "404" or "not found" must appear somewhere on the page.	3
Cosmetic	An error that does not affect the functionality of the website, such as a typo, small formatting issues, bits of visible HTML code, etc.	24
Language Error	An inability to encode or correctly convert characters between languages, often resulting in incorrect characters on the page.	2
CSS Error	An error in loading the stylesheet between the *current* and *next* pages.	≤ 1
Code on the Screen	Any error that results in non-HTML, non-SQL program code appearing on screen, including any error referring to a line number.	24
Error Message / Other Error	Either any error message, or any error that cannot be classified in any other category.	52
Form Error	Missing, malformed, or extra buttons, form fields, drop-down menus, etc, including incorrectly validating forms.	7
Missing Information	Any part of a webpage that is missing, not including images.	13
Wrong Page / No Redirect	An unexpected page is loaded.	12
Authentication	Any errors that occur during login.	6
Permission	Any errors occurring with respect to user permissions in an application, such as access being incorrectly denied to a user.	4
Session	An unexpected session timeout or other session-related issues.	1
Search	Errors occurring during searching, such as incorrectly printing out results.	2
Database	Any errors associated with accessing or querying a database, including visible SQL code being displayed.	9
Failed Upload	An error during the upload of an item.	5
Missing Image	A missing image.	3

ANOVAs for fault severity models

Feature	Correlation	F	p value
Code on the Screen	+	19.47	< 0.001
Cosmetic	-	13.23	< 0.001
Database	+	12.36	< 0.001
Authentication	+	6.99	0.01
Functional Display	-	6.00	0.01
Code Error	+	4.40	0.03

Feature	Correlation	F	p value
Cosmetic	-	30.51	< 0.001
Functional Display	+	27.12	0.01
Code on the Screen	+	22.83	< 0.001
Code Error	+	5.32	0.02
Wrong Page	-	5.31	0.02

Fault visualizations

Feature	Description
Same Page	The error is visible within the same page and application (imagine a website with frames); the title, menu, and/or sidebars stay the same
New Page	A page is loaded that does not look like other pages in the application; examples are blank pages or server-generated error messages
Generic Error Message	A human-readable wrapper around an exception, which frequently provides no useful information about the problem
Popup	The error resulted or was displayed in a popup
Server	The error was a standard server-generated complaint, such as an HTTP 404 or 500 error
stack trace	A stack trace or other visible part of non-HTML code
Other Error Message	Text exists on the page indicating there was an error (as opposed to a missing image or other "silent" fault)

Fault localizations

Cause	Description	SRCC with high severity
Database	An error in the database configuration or structure	0.66
SQL	A buggy SQL query that lead to an exception	0.69
NULL	An empty code or database object which lead to an exception	0.69
Source Code	An error due to incorrect logic in the source code	0.18
Config	Configuration settings were inconsistent	0.68
Component	A third party component was incompatible or caused an error	0.62
Upgrade	A file was missing, or a recent upgrade caused an error	0.63
Permission	The operating system failed to allocate resources or open files	0.68
Server	Incorrectly configured server	0.68

LookCost/MissCost

- Previous work uses 0.023 from the domain of bug triage
- LookCost is typically a few minutes per test case
- MissCost varies by domain (low where software can be easily updated, but high where there are high quality-of-service requirements)
- At IBM in 2008
  - LookCost is $25
  - MissCost is $450 (during QA/testing)
  - For H1, this results in a 48% reduction in cost

Future Work

- Explore ways to extend this work to other technologies
  - Asynchronous javascript
    - Automated ways of running test suites without relying on capture-replay
- Expand consumer-perceived fault severity to other domains
  - GUIs and human-computer interaction
    - Add new domain-specific features to the model
- Combine machine learning with brain imaging
  - To train classifiers to identify patterns of thought
    - Learn about the role of various brain structures in aging and memory