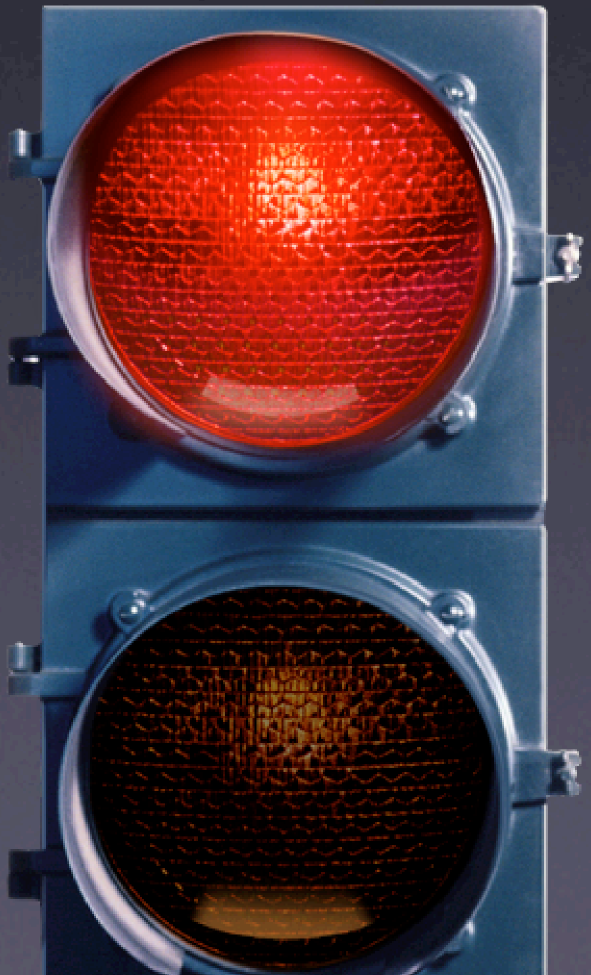
Making Programs Fail

Andreas Zeller



Two Views of Testing

- Testing means to execute a program with the intent to make it fail.
- Testing for validation:
  Finding *unknown* failures (classical view)
- Testing for debugging:
  Finding a *specific* failure (our focus)

Tests in Debugging

- Write a test to *reproduce* the problem
- Write a test to *simplify* the problem
- Run a test to *observe* the run
- Run a test to *validate a fix*
- Re-run tests to protect against *regression*

Automated Tests

- Allow for *reuse* of tests
- Allow tests that are difficult to carry out manually
- Make tests repeatable
- Increase confidence in software

Automated Tests

- Allow to isolate and simplify
    - *failure-inducing input*
    - *failure-inducing code changes*
    - *failure-inducing thread schedules*
    - *failure-inducing program state*
- More on this in the weeks to come

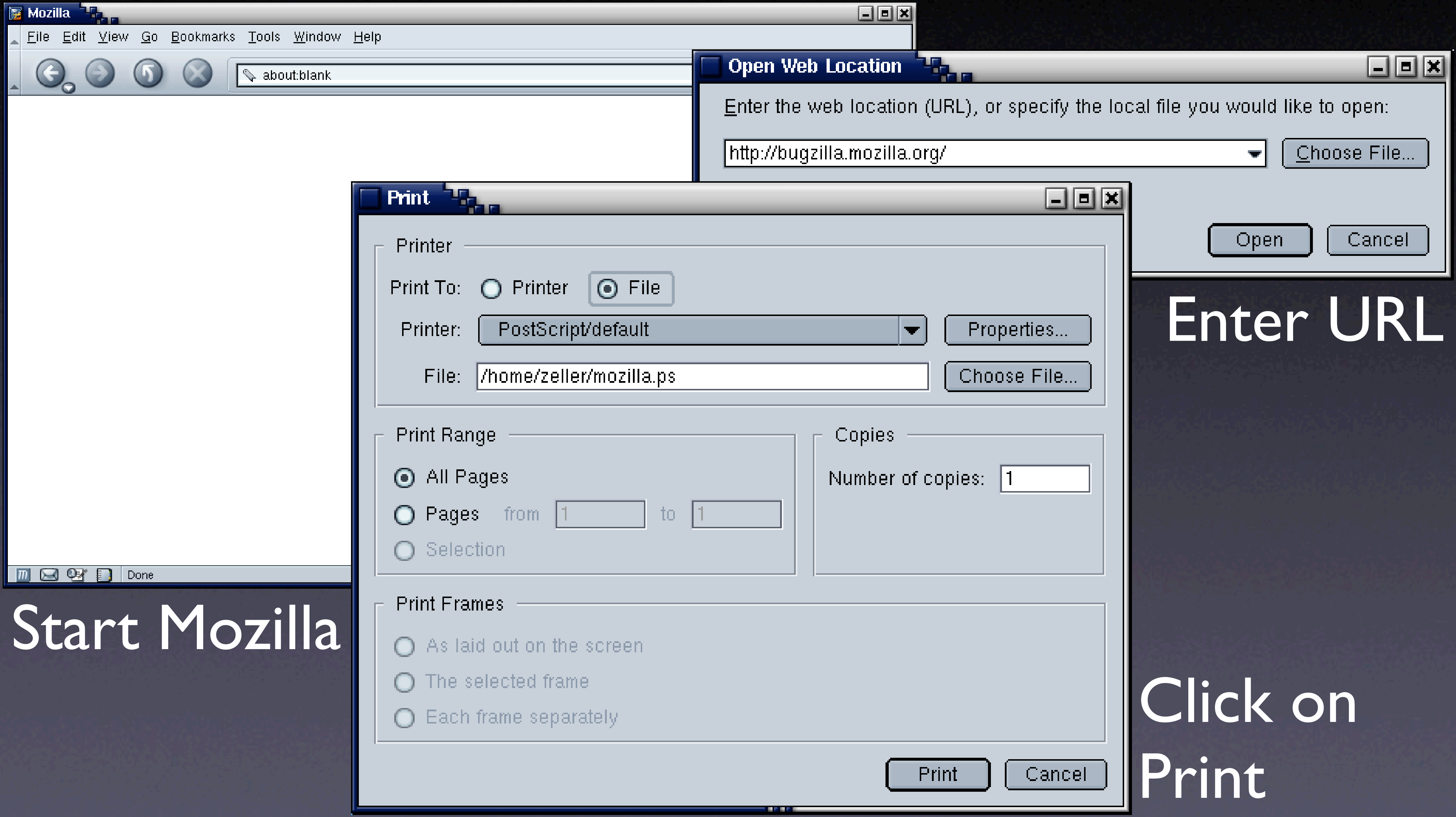
Mozilla Bug #24735

Ok the following operations cause mozilla to crash consistently on my machine

-> Start mozilla
-> Go to bugzilla.mozilla.org
-> Select search for bug
-> Print to file setting the bottom and right margins to .50 (I use the file /var/tmp/netscape.ps)
-> Once it's done printing do the exact same thing again on the same file (/var/tmp/netscape.ps)
-> This causes the browser to crash with a segfault

How do we automate this?

Simulating Interaction



about:blank

Done

Enter the web location (URL), or specify the local file you would like to open:

http://bugzilla.mozilla.org/

Printer

Print To: ○ Printer ● File

Printer: PostScript/default

File: /home/zeller/mozilla.ps

Print Range

● All Pages

○ Pages from 1 to 1

○ Selection

Copies

Number of copies: 1

Print Frames

○ As laid out on the screen

○ The selected frame

○ Each frame separately

Start Mozilla

Enter URL

Click on Print

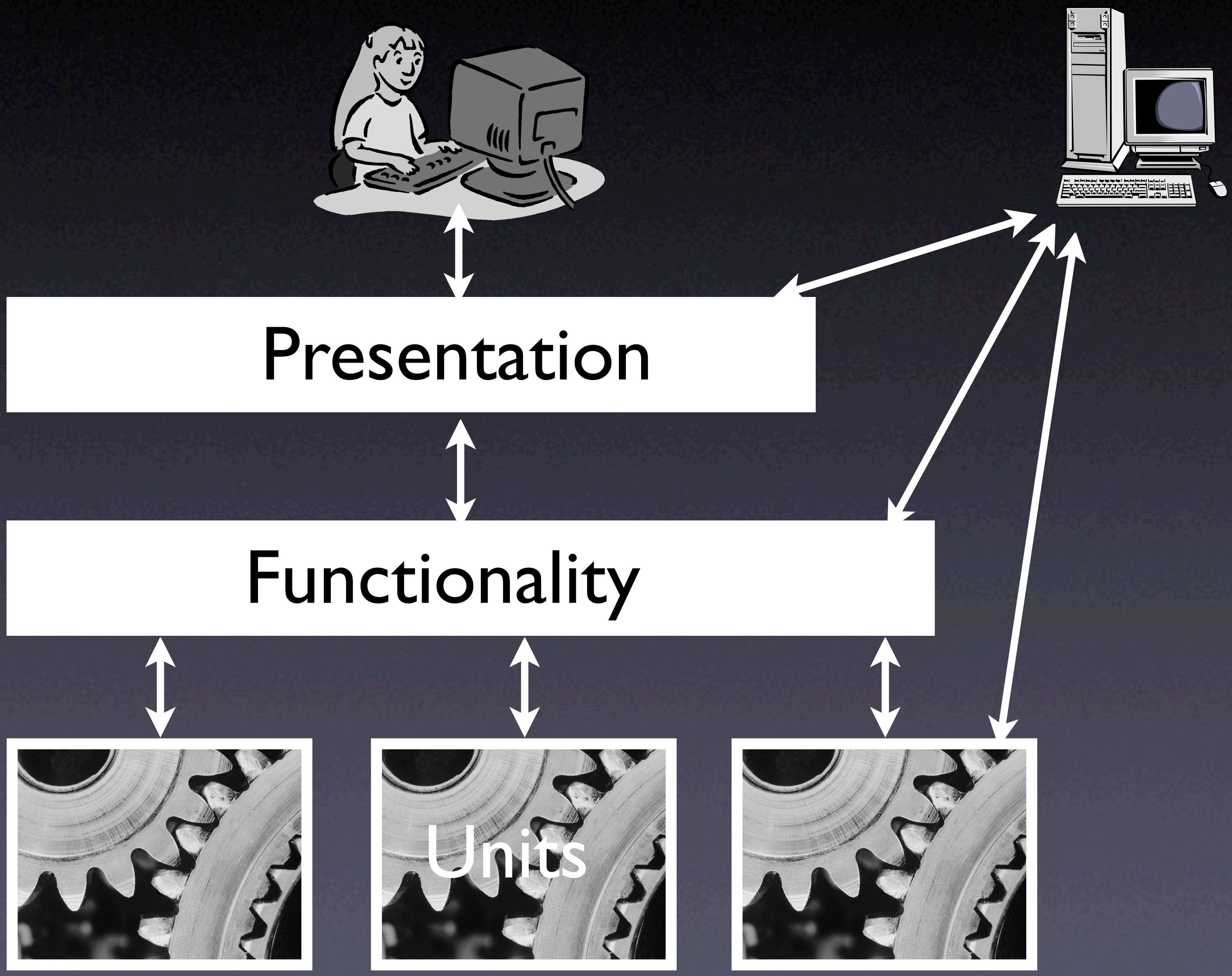
Challenges

- *Synchronization*: How do we know a window has popped up such that we can click into it?
- *Abstraction*: How do we know it's the right window?
- *Portability*: What happens on a display with different resolution or window placement?

Interaction Layers

- The *presentation layer* handles interaction with the user (generally: the environment)
- The *functionality* layer encapsulates the functionality (independent from a specific presentation)
- The *unit layer* splits functionality across cooperating units

Control Layers



Presentation

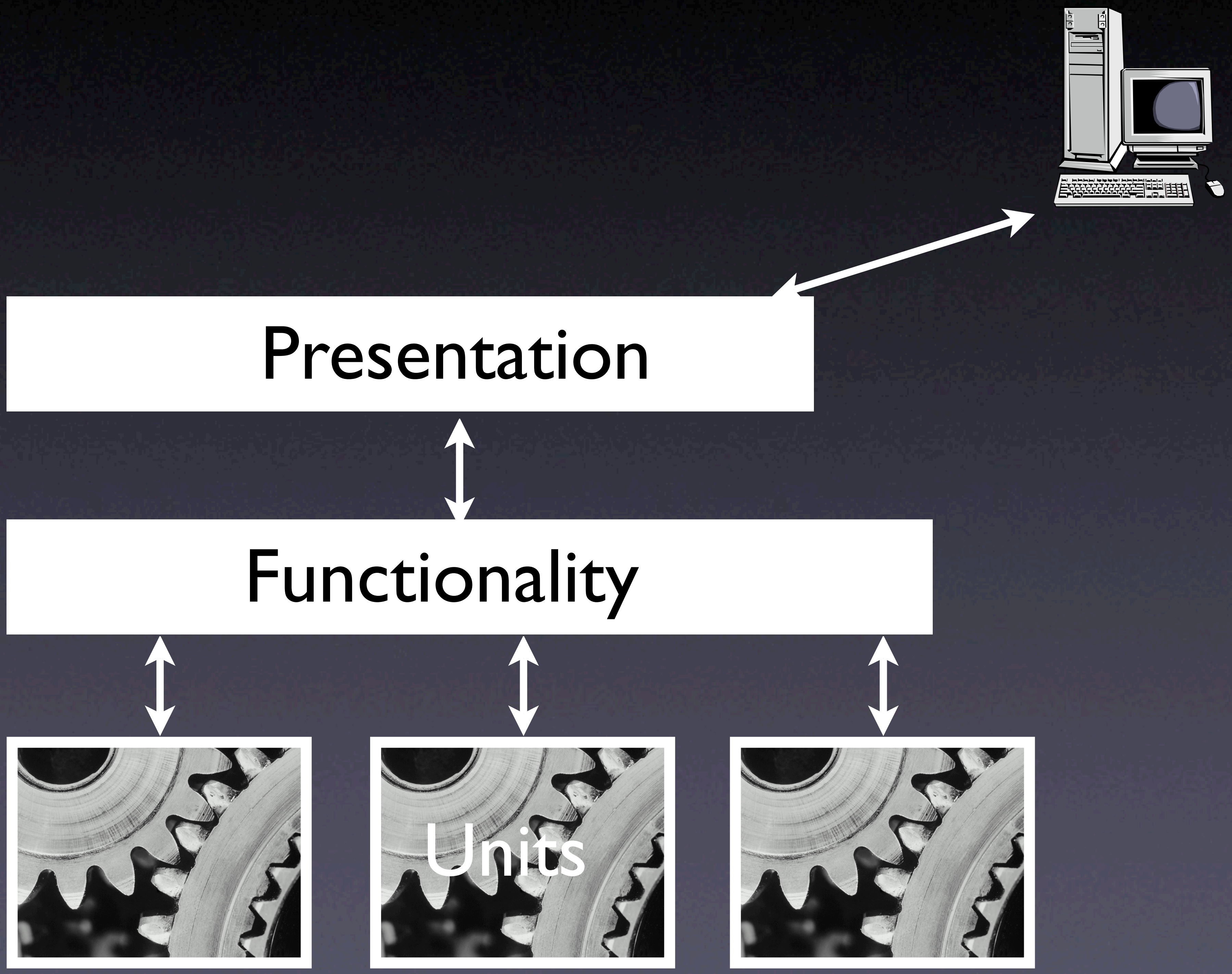
Functionality

Units

Assessing Layers

- Ease of execution. How easy is it to get control over program execution?
- Ease of interaction. How easy is it to interact with the program?
- Ease of result assessment. How can we check results against expectations?
- Lifetime of test case. How robust is my test when it comes to program changes?

Presentation Layer



Presentation

Functionality

Units

Presentation Layer

- Low-level: expressing interaction by means of mouse and keyboard events
    - Also applicable at the system level
- High-level: expressing interaction using graphical controls

Low Level Interaction

```
# 1. Launch mozilla and wait for 2 seconds
exec mozilla &
send_xevents wait 2000

# 2. Open URL dialog (Shift+Control+L)
send_xevents keydn Control_L
send_xevents keydn Shift_L
send_xevents key L
send_xevents keyup Shift_L
send_xevents keyup Control_L
send_xevents wait 500

# 3. Load bugzilla.mozilla.org and wait for 5 seconds
send_xevents @400,100
send_xevents type {http://bugzilla.mozilla.org}
send_xevents key Return
send_xevents wait 5000
```

Low Level Interaction

- Scripts can easily be *recorded*
- Scripts are *write-only*
  (= impossible to maintain)
- Scripts are *fragile*
  (= must be remade after trivial changes)

System Level Interaction

```
# Power on the machine and wait for 5s
power <= true; wait for 5000;

# Click mouse button 1
m_b1 <= true; wait for 300; m_b1 <= false;

# Click the CDROM change button
cdctrl'shortcut_out_add("/cdrom%change/...");
```

System Level Interaction

- Complete control over machine
- Good for testing and debugging system properties
- Difficult to use for application programs

Higher Level Interaction

```
-- 1. Activate mozilla
tell application "mozilla" to activate

-- 2. Open URL dialog via menu
tell application "System Events" to ¬
  tell process "mozilla" to ¬
    tell menu bar 1 to ¬
      tell menu bar item "File" to ¬
        click menu item "Open Web Location"

-- 3. Load bugzilla.mozilla.org and wait for 5 seconds
tell window "Open Web Location"
  tell sheet 1 to ¬
    set value of text field 1 to "http://bugzilla.mozilla.org/"
  click button 1
end tell
delay 5
```

Higher Level Interaction

- Scripts reference GUI elements by *name* and *numbers* (rather than coordinates)
- Much more robust against size and position changes
- But still fragile against layout changes and renamings

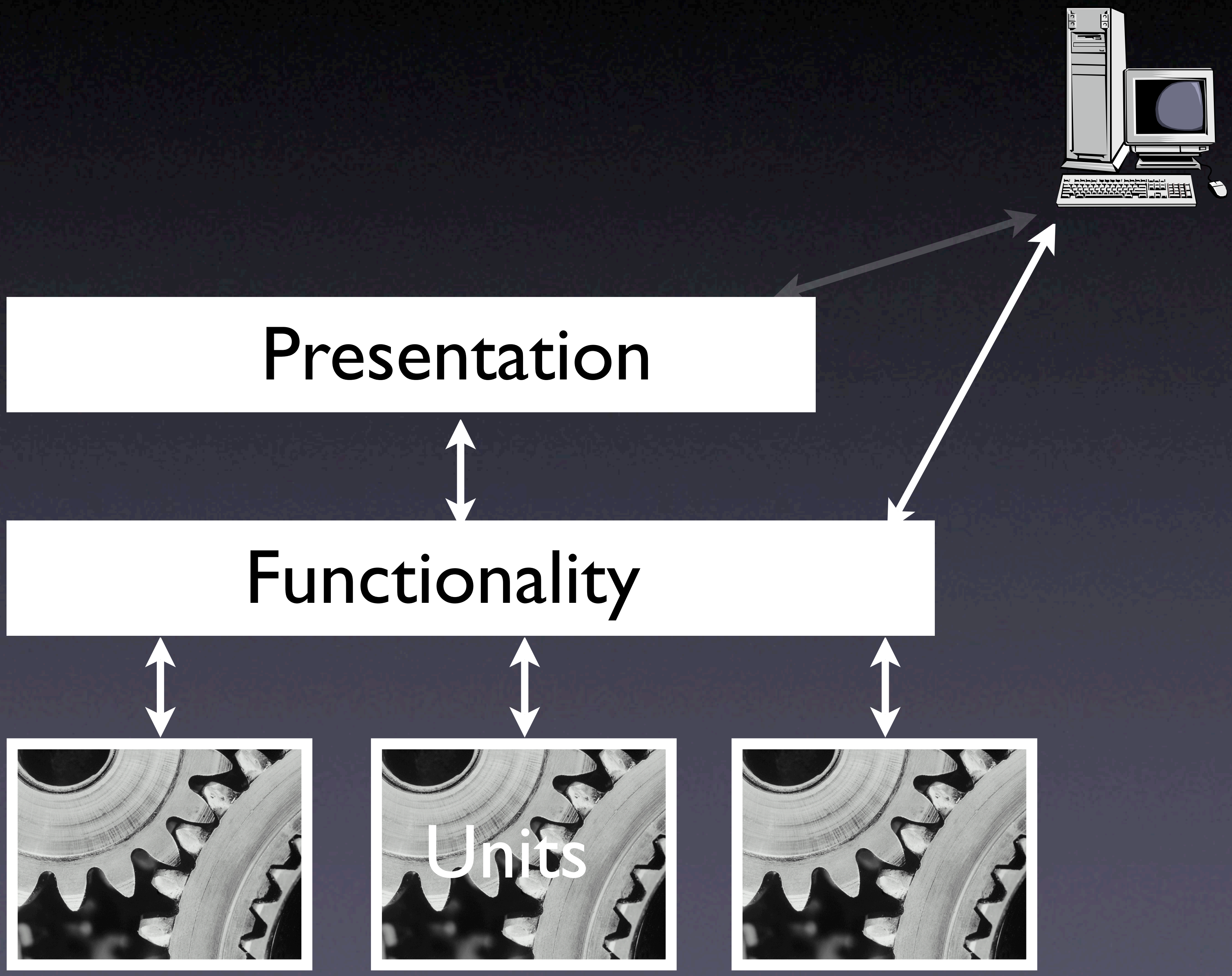
Dealing with Output

- We must be able to detect *output*
  - for *synchronization* (“is the dialog there?”)
  - for *assessment of results* (“was the test successful?”)
- Issue at entire presentation layer (low level, system level, and high level interface)

Presentation Layer

- Automation is *always feasible*
- Scripts are more or less *fragile*
- Dealing with output is greatest weakness

Functionality Layer



Presentation

Functionality

Units

Design for Automation

- Each application comes with an API for a scripting language

Check state
of application

```
tell application "Safari"
  activate
  if not (exists document 1)
    make new document at the beginning of documents
  end if
  set the URL of the front document ¬
    to "http://bugzilla.mozilla.org/"
  delay 5
end tell
```

Windows Scripting

- Most operating systems provide their own scripting language

```
' Load document
Set IE = CreateObject("InternetExplorer.Application")
IE.navigate "http://bugzilla.mozilla.org/"
IE.visible=1

' Wait until the page is loaded
While IE.Busy
    WScript.Sleep 100
Wend
```

Emacs Scripting

- Some applications are built around a script interpreter

```
(defun ispell-toggle ()
  "Toggle ispell dictionary between english and german"
  (interactive)
  (cond ((equal ispell-local-dictionary nil)
        (ispell-change-dictionary "american"))
      ((equal ispell-local-dictionary "deutsch8")
       (ispell-change-dictionary "american"))
      (t
       (ispell-change-dictionary "deutsch8")))
  (ispell-init-process)
  (message (concat "Using " ispell-local-dictionary
            "ispell dictionary")))
```

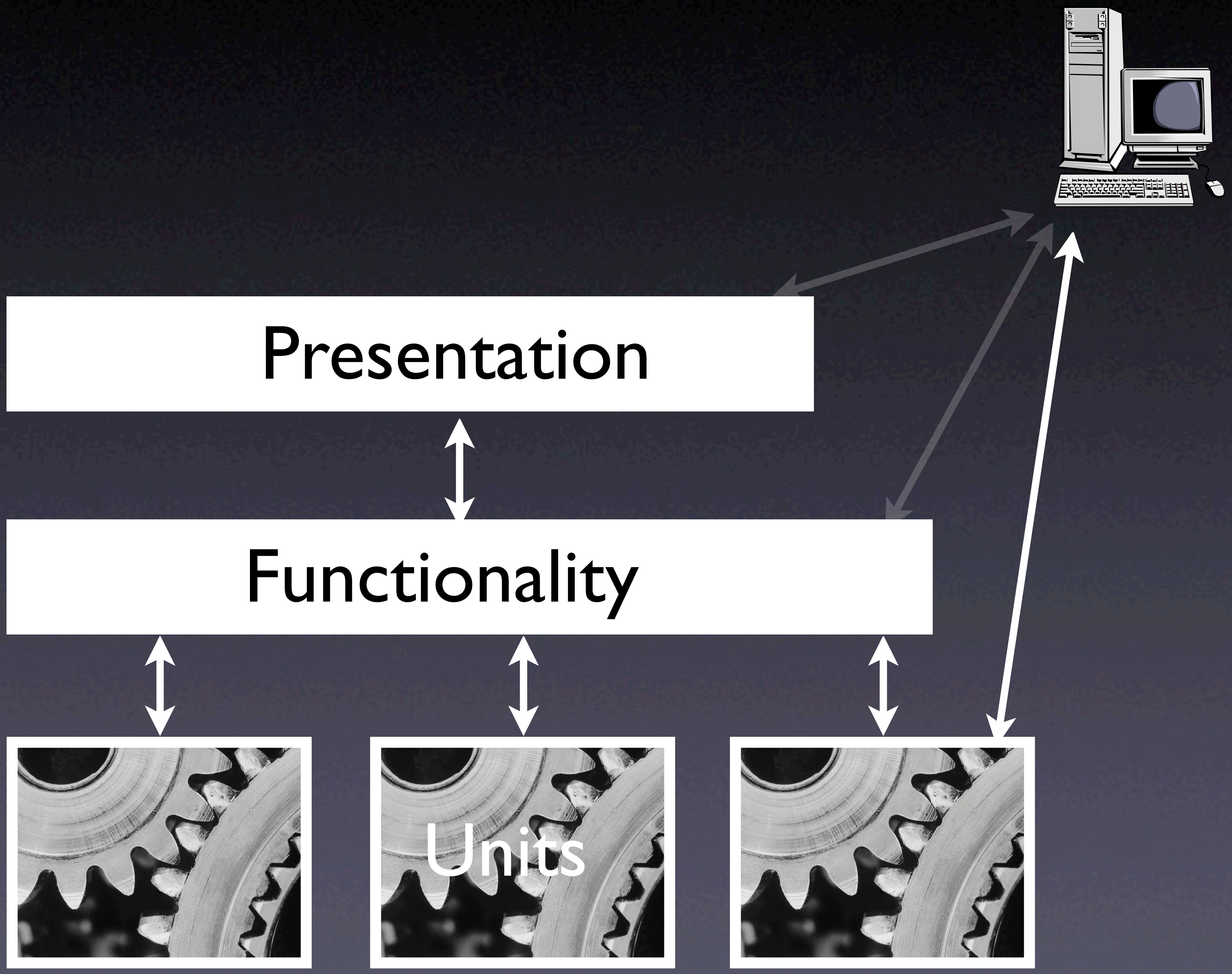
Scripting Languages

- OS-specific languages (MacOS, Windows)
- Perl, Python, Tcl
- Lisp, Scheme, Guile
- Command-line languages (Unix shell)
- Component languages (.NET, Corba)
- … or roll your own (but beware!)

Functionality Layer

- Results can be easily assessed
- Scripts are robust against changes (as long as automation interface remains stable)
- Requires clear separation between presentation and functionality

Unit Layer



Presentation

Functionality

Units

Unit Tests

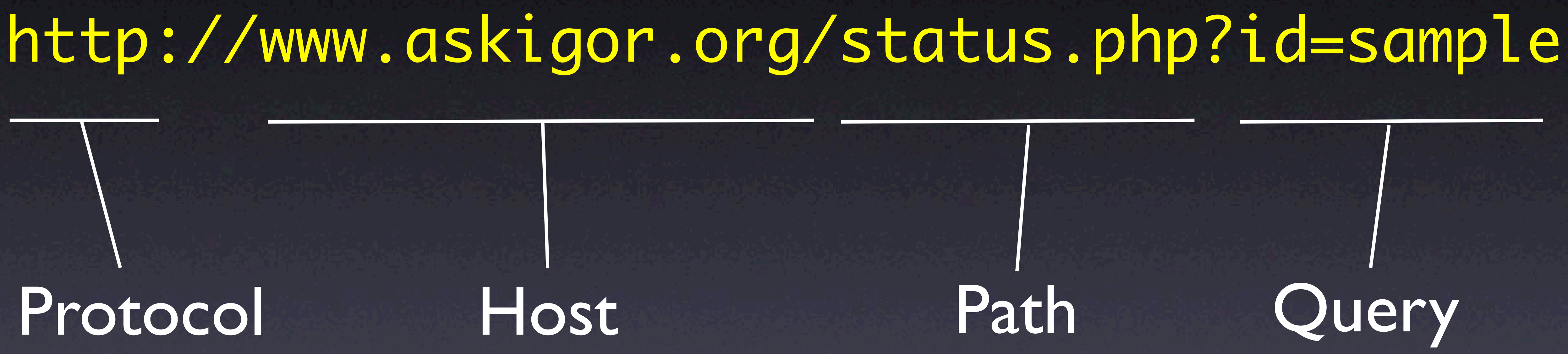
- Directly access units (= classes, modules, components…) at their programming interfaces
- Encapsulate a set of tests as a single syntactical unit
- Available for all programming languages (JUNIT for Java, CPPUNIT for C++, etc.)

Running a Test

A test case…

1. *sets up an environment for the test*
2. *tests* the unit
3. *tears down* the environment again.

Testing a URL Class



`http://www.askigor.org/status.php?id=sample`

Protocol

Host

Path

Query

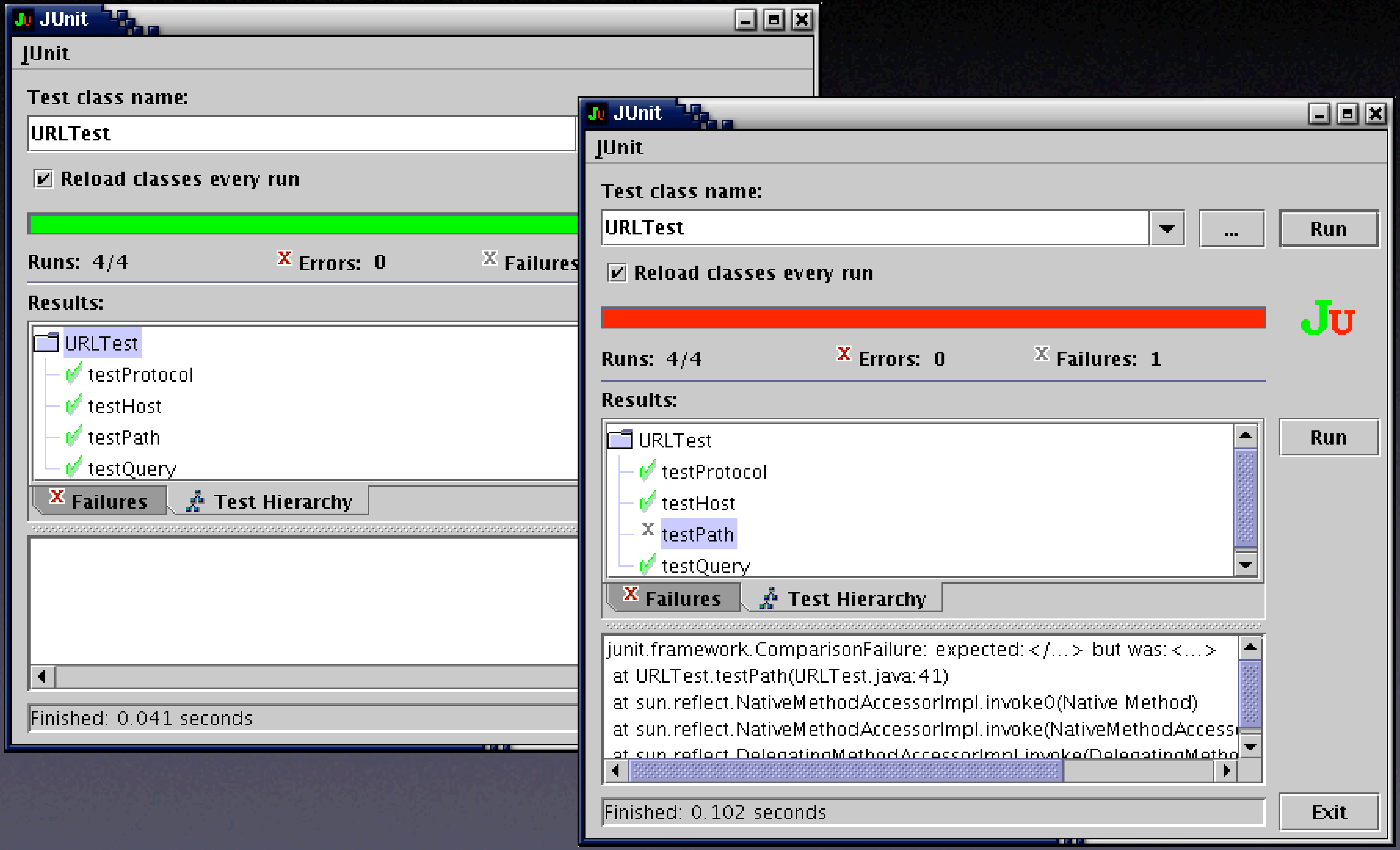
```
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class URLTest extends TestCase {
    private URL askigor_url;

    // Create new test
    public URLTest(String name) { super(name); }

    // Assign a name to this test case
    public String toString() { return getName(); }

    // Setup environment
    protected void setUp() {
        askigor_url = new URL("http://www.askigor.org/" +
                              "status.php?id=sample"); }
    // Release environment
    protected void tearDown() { askigor_url = null;}
```

```
// Test for protocol (http, ftp, etc.)
public void testProtocol() {
  assertEquals(askigor_url.getProtocol(), "http");
}

// Test for host
public void testHost() {
  int noPort = -1;
  assertEquals(askigor_url.getHost(), "www.askigor.org");
  assertEquals(askigor_url.getPort(), noPort);
}

// Test for path
public void testPath() {
  assertEquals(askigor_url.getPath(), "/status.php");
}

// Test for query part
public void testQuery() {
  assertEquals(askigor_url.getQuery(), "id=sample");
}
```

The test case can be used as a *specification!*

```
    // Set up a suite of tests
    public static Test suite() {
        TestSuite suite = new TestSuite(URLTest.class);
        return suite;
    }

    // Main method: Invokes GUI
    public static void main(String args[]) {
        String[] testCaseName =
            { URLTest.class.getName() };
        // junit.textui.TestRunner.main(testCaseName);
        junit.swingui.TestRunner.main(testCaseName);
        // junit.awtui.TestRunner.main(testCaseName);
    }
}
```

JUnit



Test class name: URLTest

☑ Reload classes every run

Runs: 4/4 Errors: 0 Failures

Results:

- URLTest
  - ✓ testProtocol
  - ✓ testHost
  - ✓ testPath
  - ✓ testQuery

Finished: 0.041 seconds

Test class name: URLTest

☑ Reload classes every run

Runs: 4/4 Errors: 0 Failures: 1

Results:

- URLTest
  - ✓ testProtocol
  - ✓ testHost
  - ✗ testPath
  - ✓ testQuery

```
junit.framework.ComparisonFailure: expected:</...> but was:<...>
 at URLTest.testPath(URLTest.java:41)
 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
 at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccess
 at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMetho
```

Finished: 0.102 seconds

PyUnit

- Unit testing framework for Python
- Simple variant: just overload runTest()

```
import unittest

class DefaultWidgetSizeTestCase(unittest.TestCase):
    def runTest(self):
        widget = Widget("The widget")
        assert widget.size() == (50,50), \
               'incorrect default size'
```

PyUnit Fixtures

```
class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget("The widget")
    def tearDown(self):
        self.widget.dispose()
        self.widget = None
    def testDefaultSize(self):
        assert self.widget.size() == (50,50), \
             'incorrect default size'
    def testResize(self):
        self.widget.resize(100,150)
        assert self.widget.size() == (100,150), \
            'wrong size after resize'
```

Running PyUnit tests

```
if __name__ == “__main__”:
    unittest.main()
```

```
$ python unittest.py widgettests.WidgetTestSuite
```

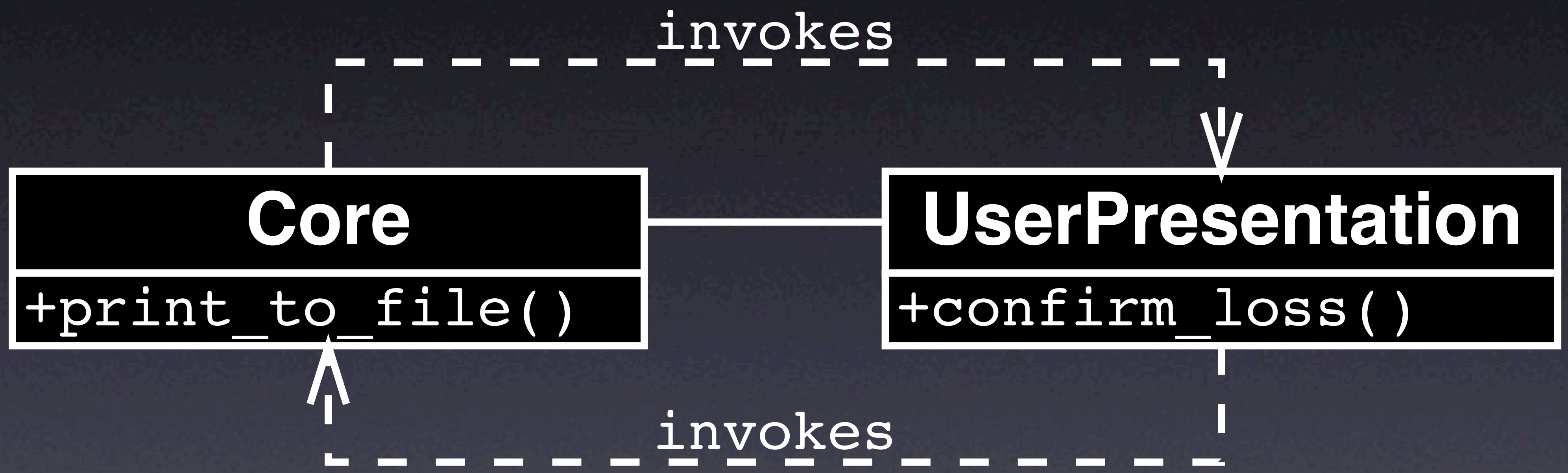
http://pyunit.sourceforge.net/pyunit.html

Isolating Units

- How do we deal with classes that depend on others?

```
void print_to_file(string filename)
{
    if (path_exists(filename)) {
        // FILENAME exists; ask user to confirm overwrite
        bool confirmed = confirm_loss(filename);
        if (!confirmed)
            return;
    }
    // Proceed printing to FILENAME...
}
```

Circular Dependency

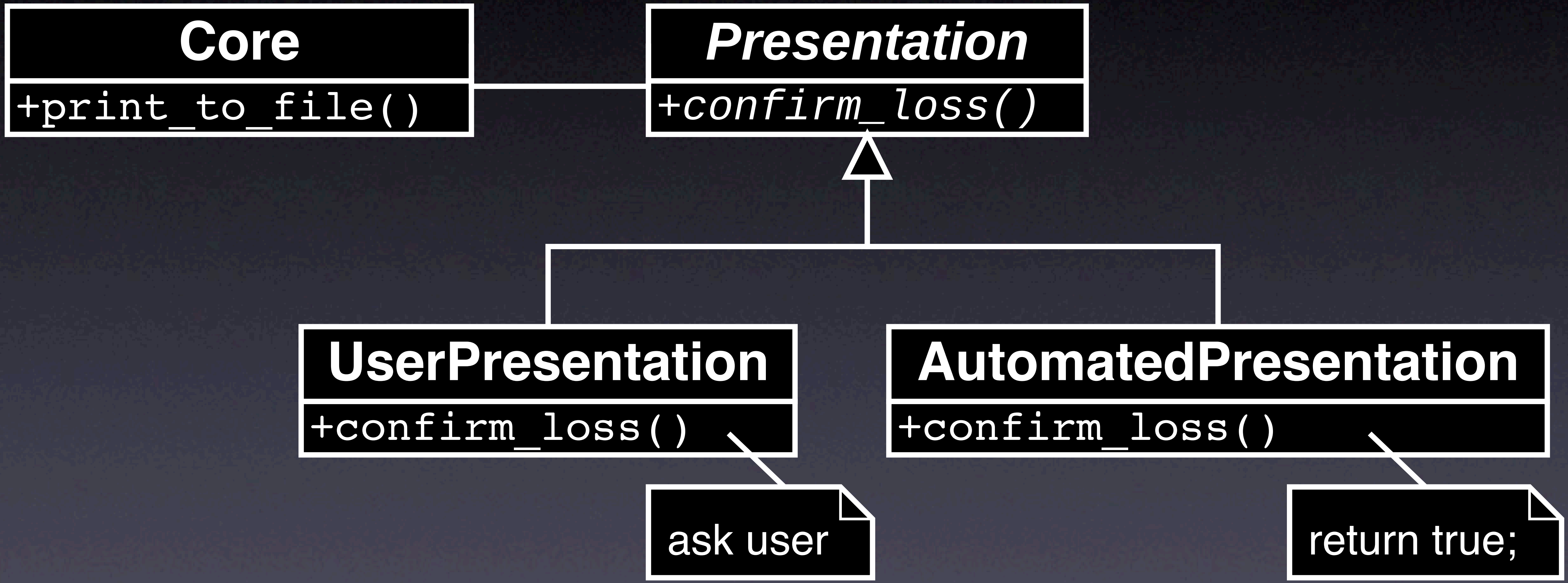


Both units depend on each other!

Broken Dependency

```
void print_to_file(string filename,
                   Presentation *presentation)
{
    if (path_exists(filename))
    {
        // FILENAME exists;
        // ask user to confirm overwrite
        bool confirmed =
            presentation->confirm_loss(filename);
        if (!confirmed)
            return;
    }

    // Proceed printing to FILENAME
    ...
}
```

Revised Dependency



Depend on abstraction rather than details!

Dependency Inversion

To break the dependency from A to B,

1. Introduce an abstract superclass B’
2. Set up A such that it depends on B’ (rather than B)
3. Introduce alternate subclasses of B’ that can be used with A

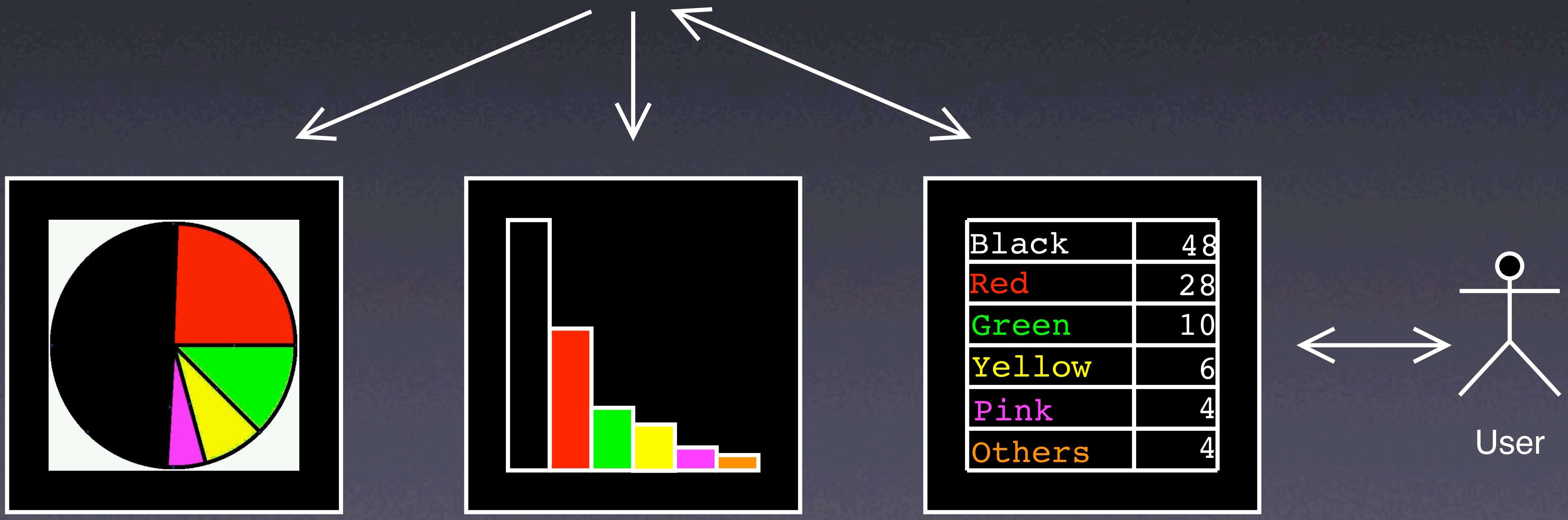
Design for Debugging

- Basic idea: decompose the system such that dependencies are minimized
- Each component depends on a minimum of other components for testing (and debugging)

Model–View–Controller

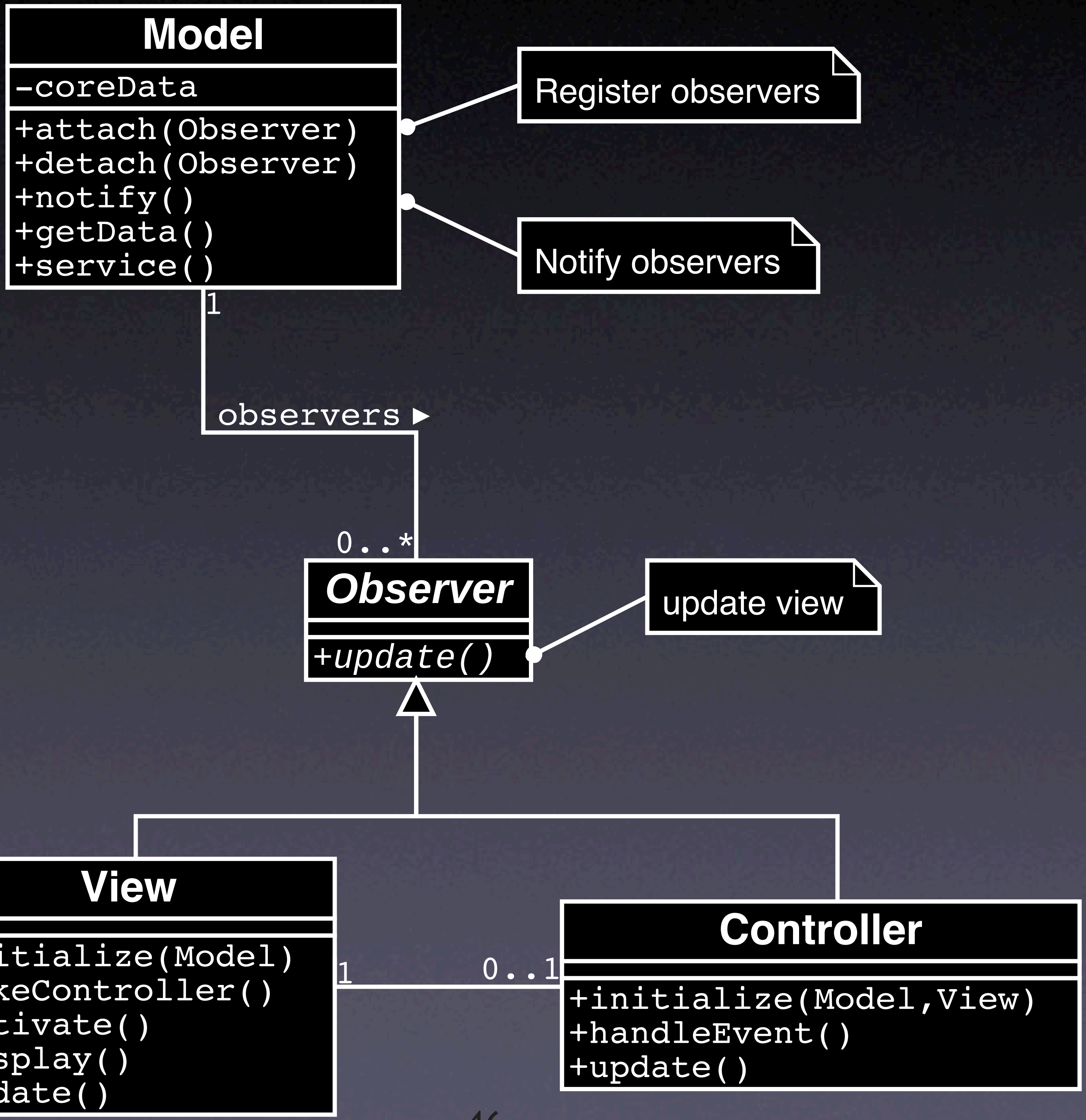
Black:	48%
Red:	28%
Green:	10%
Yellow:	6%
Pink:	4%
Others:	4%

Separate functionality and presentations



Black	48
Red	28
Green	10
Yellow	6
Pink	4
Others	4

The MVC Pattern



General Design Rules

- High cohesion. Those units that operate on common data should be grouped together.
- Low coupling. Units that do not share common data should exchange as little information as possible.

Prevent Problems

Specify	Test early	Test first
Test often	Test enough	Have reviews
Check the code	Verify	Assert

Concepts

- ★ To test for debugging, one must…
  - create a test to reproduce the problem
  - run the test several times during debugging, and
  - run the test before new releases to prevent regression
- ★ Automate as much as possible

Concepts (2)

- To test at the presentation layer, simulate human interaction
- To test at the functionality layer, use an automation interface
- To test units, use the unit API to control it and assess its results

Concepts (3)

- To isolate a unit, break dependencies using the dependency inversion principle
- To design for debugging, reduce the amount of dependencies
- A variety of techniques is available to prevent errors and problems