ASPROTUNITY
www.asprotunity.com

Design For Testability

What, Why, How

ACCU 2015 conference

Giovanni Asproni
email: gasproni@asprotunity.com
twitter: @gasproni
linkedin: http://www.linkedin.com/in/gasproni

Software Testability

"...is the degree to which a software artifact (i.e. a software system, software module, requirements- or design document) supports testing in a given test context. If the testability of the software artifact is high, then finding faults in the system (if it has any) by means of testing is easier."

http://en.wikipedia.org/wiki/Software_testability

Design For Testability

Design for Test (aka "Design for Testability" or "DFT") is a name for design techniques that add certain testability features to a microelectronic hardware product design. The premise of the added features is that they make it easier to develop and apply manufacturing tests for the designed hardware. The purpose of manufacturing tests is to validate that the product hardware contains no defects that could, otherwise, adversely affect the product's correct functioning.

http://en.wikipedia.org/wiki/Design_for_testing

Design And Architecture

"All architecture is design but not all design is architecture"

Grady Booch

http://handbookofsoftwarearchitecture.com/?p=63

Design And Architecture

Every software-intensive system has an architecture. In some cases that architecture is intentional, while in others it is accidental. Most of the time it is both, born of the consequences of a myriad of design decisions made by its architects and its developers over the lifetime of a system, from its inception through its evolution. In that sense, **the architecture of a system is the naming of the most significant design decisions that shape a system, where we measure significant by cost of change and by impact upon use**.

http://handbookofsoftwarearchitecture.com/?p=63

Architecture Is For (1/2)

- Modeling, and reasoning about, important aspects of the system

- Quality attributes: testability, scalability, performance, latency, etc., often, cannot easily be retrofitted (easily)

Architecture Is For (2/2)

- Recording and communicating decisions
- Guiding lower level design and implementation decisions
  - E.g., avoid "TDDing" the system in the wrong direction
- Splitting work among teams (Conway's Law)

Conway's Law

Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations.

http://www.melconway.com/Home/Committees_Paper.html

Testing shows the presence,
not the absence of bugs.

Edsger Dijkstra

In theory, theory and practice are the same.
In practice, they are not.

Albert Einstein

Tests are mainly for maintainability and evolution.

Well Chosen Tests

- Give us confidence changes don't break existing functionality
- Scale much better than formal methods and inspections
  - Albeit formal methods and (formal) inspections might catch more defects
  - Most tests can be automated

Cost Of Software

- $cost_{total} = cost_{develop} + cost_{maintain}$

- $cost_{maintain} = cost_{understand} + cost_{change} + cost_{test} + cost_{deploy}$

- The above costs are related with each other

Cost Of Maintenance

- Usually $cost_{maintain}/cost_{total}$ ~ 60%
- It is important to write maintainable software

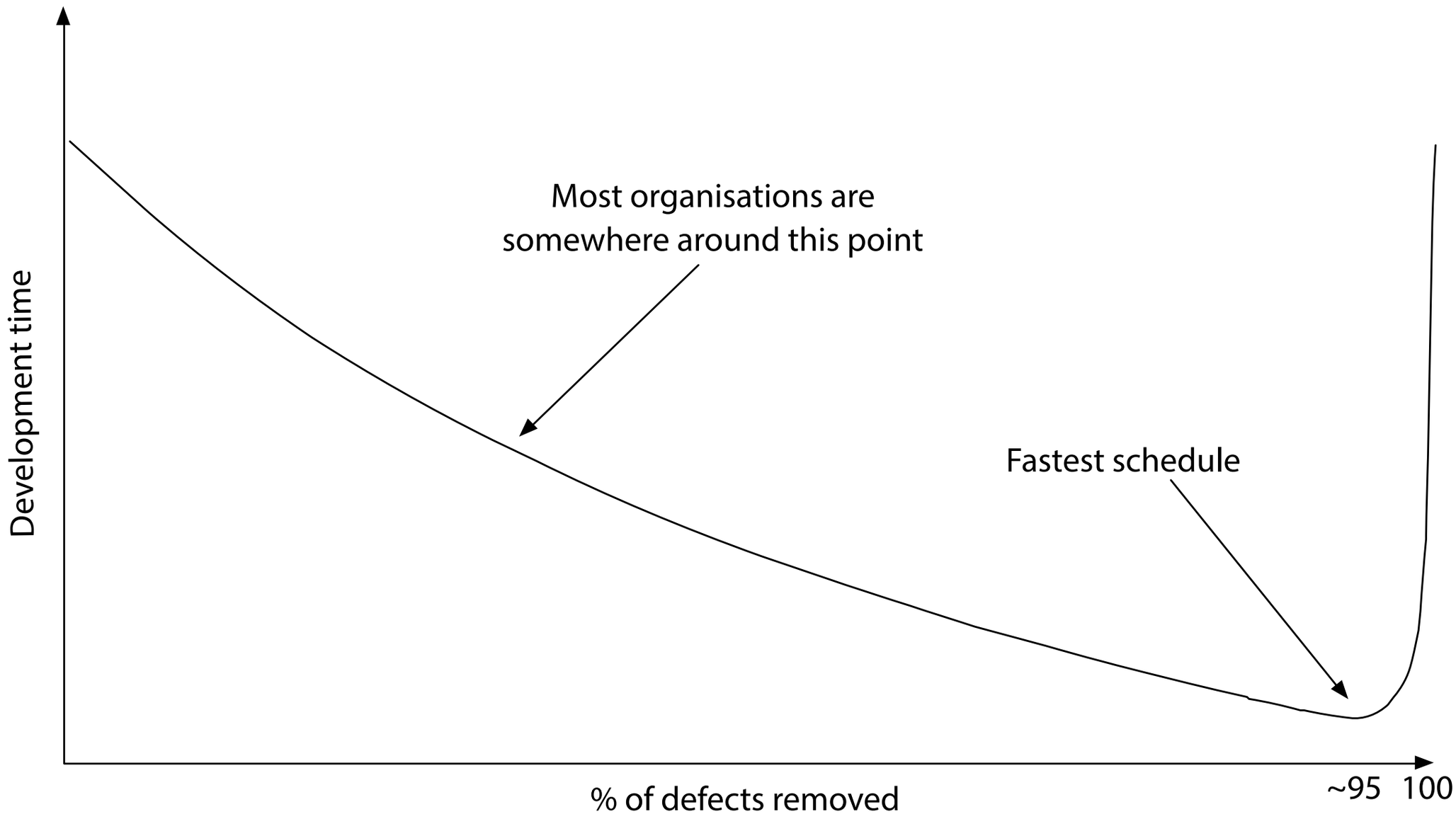
"Frequently Forgotten Fundamental Facts about Software Engineering" Robert Glass, IEEE Software, May/June 2001, http://www.eng.auburn.edu/~hendrix/comp6710/readings/Forgotten_Fundamentals_IEEE_Software_May_2001.pdf

How Quality Affects Costs

As of 2011, the **average cost per function** point in the United States is about **$1,000 to build** software applications and another **$1,000 to maintain** and support them for five years: **$2,000 per function point in total**. For projects that **use effective combinations of defect prevention and defect removal** activities and achieve high quality levels, average development costs are only about **$700 per function point** and maintenance, and support costs drop to about **$500 per function point**: **$1,200 per function point in total**.

From “The Economics of Software Quality”,
Capers Jones and Olivier Bonsignour

How Quality Affects Delivery Time



Source Steve McConnell: http://www.stevemcconnell.com/articles/art04.htm

"Quality is an accelerator"

John Clifford, Construx Software

"To be tested a system has to be designed to be tested"

From "The Art Of System Architecting"

Implications On Design

- Added complexity on some dimensions
  - But added simplicity in others
- More stuff to take care of
  - Tests need to be maintained
  - ...But it's still cheaper than running the application manually to smoke test changes...

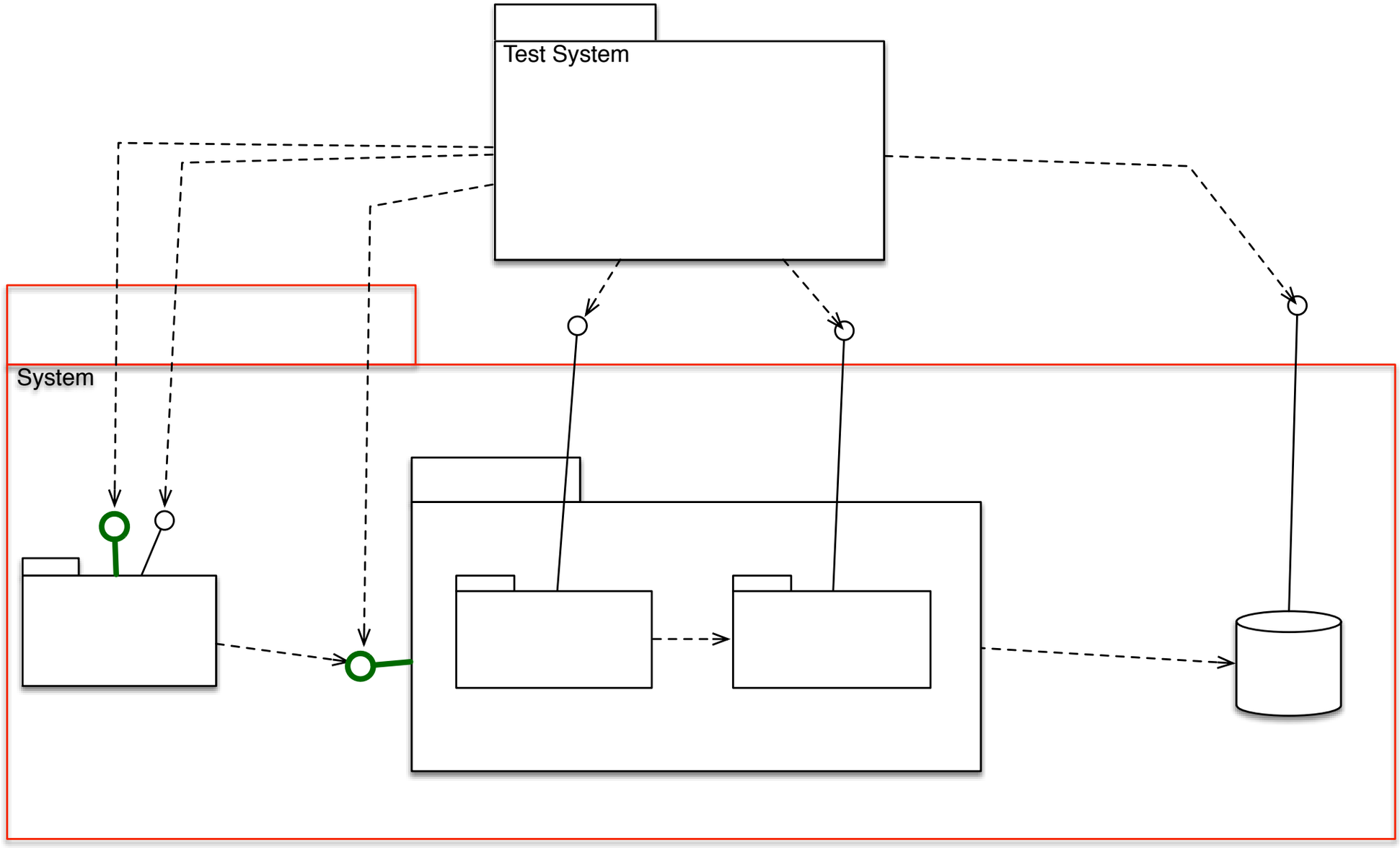
Hooks

- Dependency injection (technique NOT frameworks)
- At the language level
- Via configuration

Hooks

- APIs for testing
  - To query / change the state of the system
  - To query / change the configuration of the system
  - Notifications
  - Logs
- They often become support APIs

```
@Test
public void generatesProxyCorrectly() {

    final int value = 10;
    final String expectedMethodName = ServiceInterface.class.getMethods()[0]
            .getName();
    final Object[] args = {value};

    serviceCaller = context.mock(ServiceCaller.class);
    proxyMaker = new ServiceProxyMaker(serviceCaller);

    context.checking(new Expectations() {{
        oneOf(serviceCaller).call(args, expectedMethodName, serviceAddress, Void.TYPE);
        will(returnValue(null));
    }});

    ServiceInterface generatedProxy = proxyMaker.make(serviceAddress,
            ServiceInterface.class).service();

    generatedProxy.call(value);

    context.assertIsSatisfied();
}
```



Entry points for standard users

Entry points for testing

Hooks And Runtimes

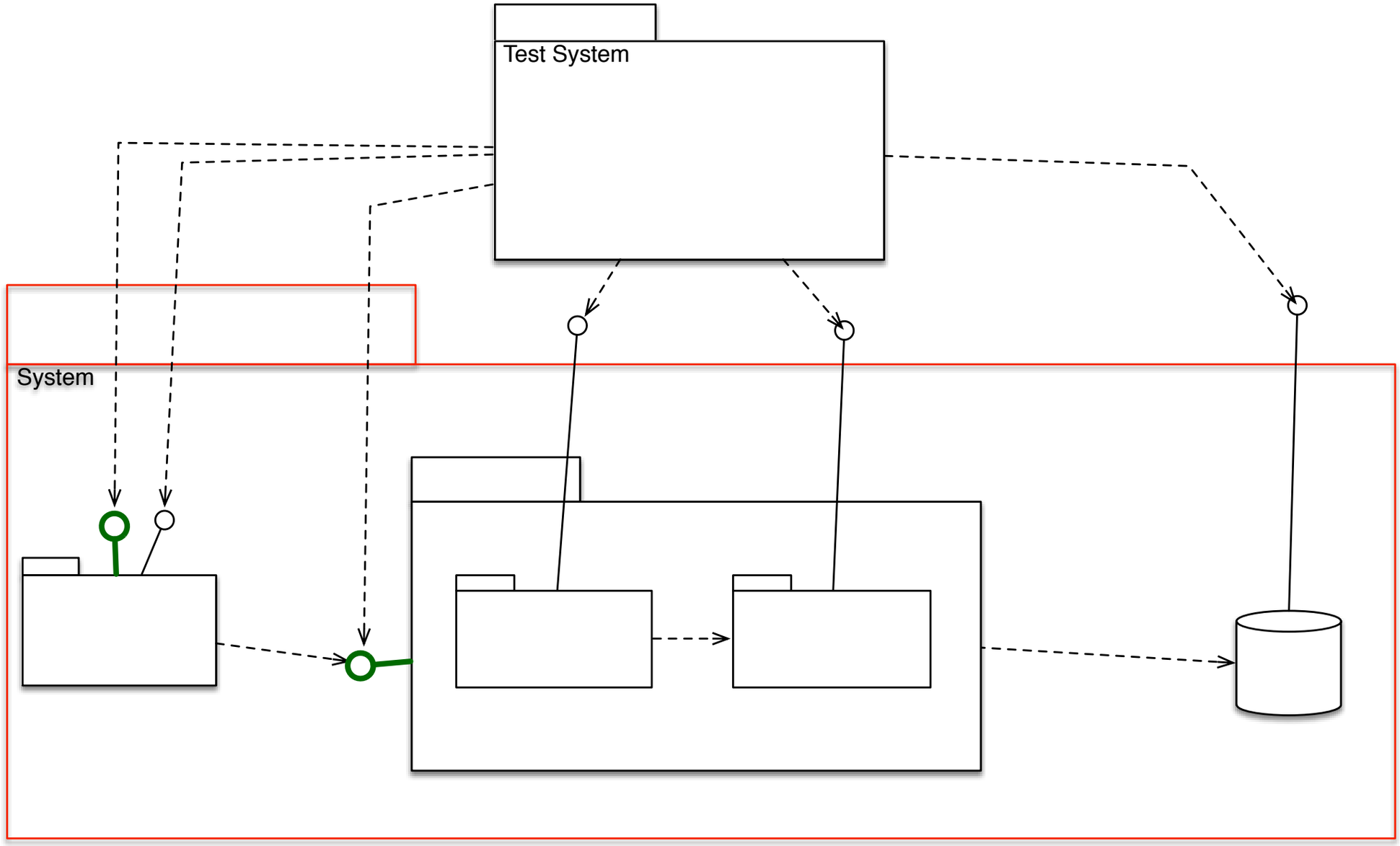
- Tests and system (partially) in same runtime
  - Allows the use of dependency injection at the language level
- Tests and system in separate runtimes
  - Dependencies are setup via configuration
  - May rely more on special APIs for testing

```
public class AuctionSniperEndToEndTest {
  private final FakeAuctionServer auction = new FakeAuctionServer("item-54321");
  private final ApplicationRunner application = new ApplicationRunner();

  @Test public void sniperJoinsAuctionUntilAuctionCloses() throws Exception {
    auction.startSellingItem();                  // Step 1
    application.startBiddingIn(auction);         // Step 2
    auction.hasReceivedJoinRequestFromSniper(); // Step 3
    auction.announceClosed();                    // Step 4
    application.showsSniperHasLostAuction();     // Step 5
  }

  // Additional cleanup
  @After public void stopAuction() {
    auction.stop();
  }
  @After public void stopApplication() {
    application.stop();
  }
}
```

From: "Growing Object Oriented Software Guided By Tests"

Same Runtime: Manage Global State Carefully

- Global application state should not spill over
- Frameworks must play nicely
- Avoid singletons and global variables



Entry points for standard users

Entry points for testing

Same Or Separate Runtimes?

- Same runtimes
  - More fine grained functional testing
  - Easier TDD at all levels
- Separate runtimes
  - Better for testing qualities (aka “non-functionals”)
  - Proper full system testing

Not Only About Tests And Code

- Ease of building the system
- Deployability
- Configurability

Deployability

- How reliably and easily can the system be deployed in a particular environment?
  - Does it require some special hardware configuration?
  - Does it require some special OS configuration?
  - Does it require a special directory structure?
  - Etc.

Configurability

- How easily can paths, dbs, and other external resources be configured? E.g.:
  - Are databases hardcoded or not?
  - Are paths fixed or not?
  - Are external resources fixed or not?
  - Can resources be configured at compile time or runtime?
- Too few or too many (or the wrong) options could be a problem

Test First at all levels makes it easier to design a system for testability

But you will probably need also some upfront design

Design Approaches

- Big upfront design
- No upfront design
- Rough upfront design

Design Approaches (A Different Perspective)

- Architecture-indifferent design
- Architecture-focused design
- Architecture hoisting

From: "Just Enough Software Architecture", George Fairbanks

Big Upfront Design Doesn't Work

- We are unable to predict everything
- Difficult to see how testability will be affected
  - Some things that seem good in theory don't work in practice
- Prone to speculative design
  - Test unnecessary functionality

No Upfront Design May Work If

- The design "emerges" by coding and refactoring
  - Can be quite effective
    - Provided tests and code are written at the same time
- Requires a higher level of expertise
- Challenging for teams bigger than 3-4 people
  - Doesn't work for large systems

Rough Upfront Design

- Helps the team to keep the big picture in mind
  - Pull in the same direction
  - E.g., helps in choosing the right refactoring paths
- Useful to hoist important qualities
  - They need to be taken care of early
- Scope for changes in design to improve testability

Ilities

- They are also known as **non-functional requirements** or **qualities**
  - Scalability
  - Security
  - Performance
  - Usability
  - Etc.

All quality requirements should be quantified.

Tom Gilb, ACCU 2010 conference

Travel Light

- No speculative design
- Remove “reusable” and “flexible” from your dictionary!

Travel Light: Beware Of Frameworks

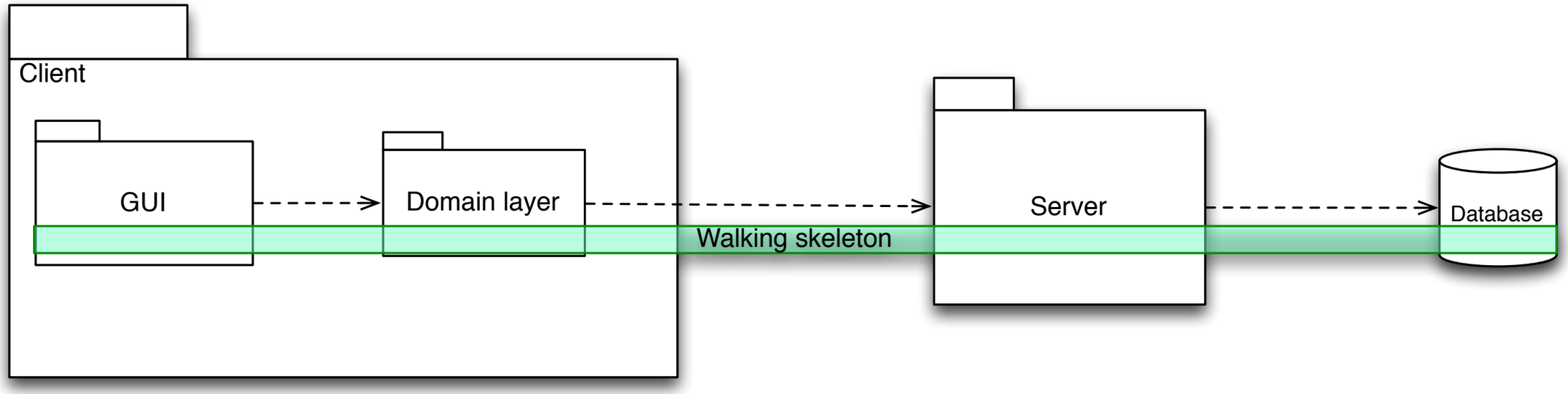
- Can be very useful...
- ...but they can impair testability

Walking Skeleton

A Walking Skeleton is a tiny implementation of the system that performs a small end-to-end function. It need not use the final architecture, but it should link together the main architectural components. The architecture and the functionality can then evolve in parallel.

From: http://alistair.cockburn.us/Walking+skeleton

Walking Skeleton



Risk Based Testing

- Least testing
  - Low likelihood of failure, low consequences of failure
- Moderate testing
  - Low likelihood of failure, high consequences of failure
  - High likelihood of failure, low consequences of failure
- Most testing
  - High likelihood of failure, High consequences of failure

From: "Perfect Software and Other Illusions about Testing", Jerry Weinberg

What Kind Of Tests?

- Manual Tests?
- System Tests?
- Component Tests?
- Unit Tests?

Tests are mainly for maintainability and evolution.

Tests Need To

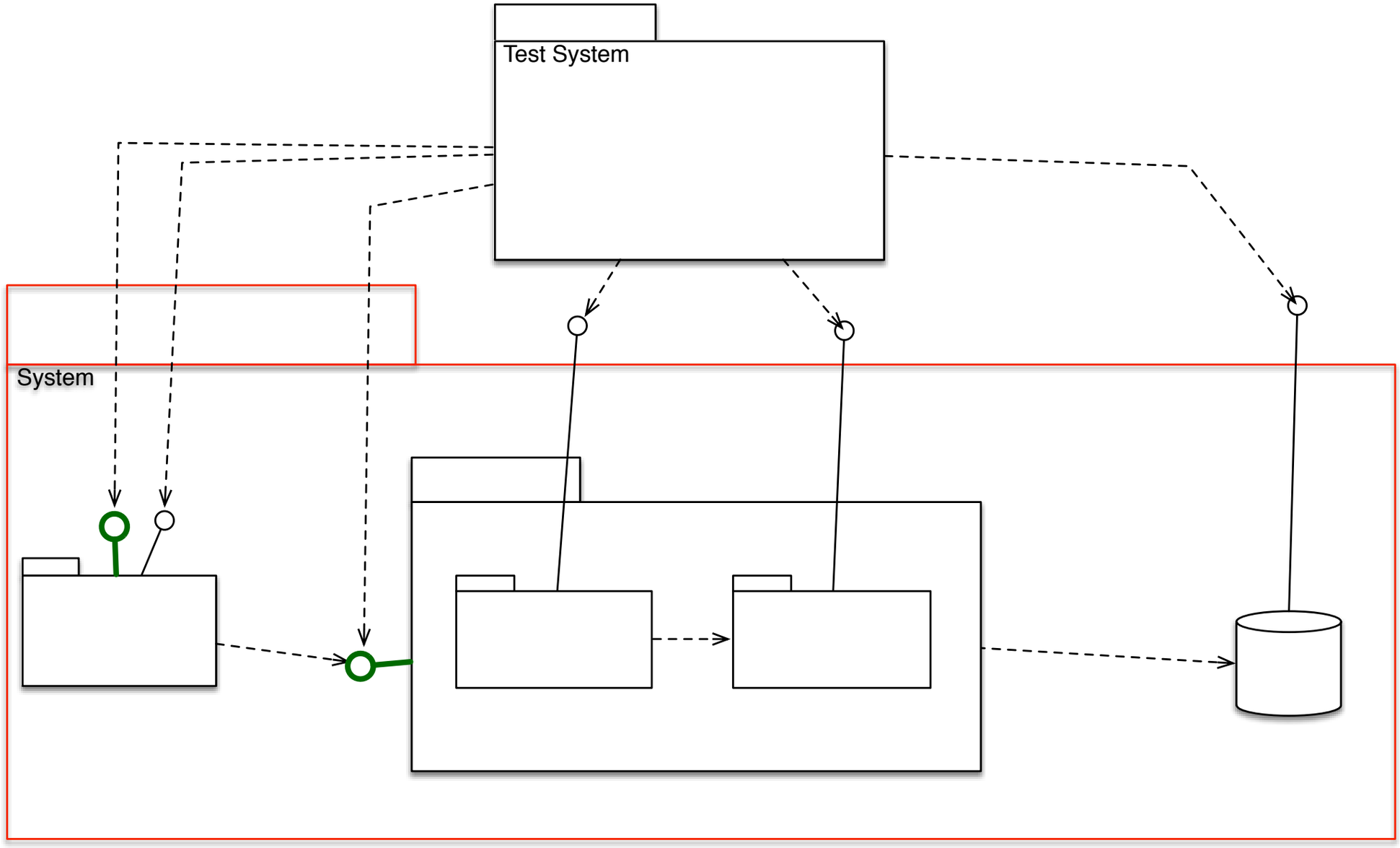
- Be reasonably simple to create
  - Or people will not create enough of them, or, perhaps, create just some easy, but not necessarily high value, ones
- Provide feedback at the right granularity level (observability and controllability)
  - Running an entire distributed system to check the implementation of a single class or method is inconvenient, to say the least
- Easy and fast to run
  - Or they won't be executed as often as they should be (if at all)

http://asprotunity.com/blog/system-tests-can-kill-your-project/

Testability Measures

- Observability: how easy it is to infer internal states from external outputs
- Controllability: how easy it is to set some internal states of the system using external inputs

```
@Test
public void generatesProxyCorrectly() {

    final int value = 10;
    final String expectedMethodName = ServiceInterface.class.getMethods()[0]
            .getName();
    final Object[] args = {value};

    serviceCaller = context.mock(ServiceCaller.class);
    proxyMaker = new ServiceProxyMaker(serviceCaller);

    context.checking(new Expectations() {{
        oneOf(serviceCaller).call(args, expectedMethodName, serviceAddress, Void.TYPE);
        will(returnValue(null));
    }});

    ServiceInterface generatedProxy = proxyMaker.make(serviceAddress,
            ServiceInterface.class).service();

    generatedProxy.call(value);

    context.assertIsSatisfied();
}
```



Entry points for standard users

Entry points for testing

System Tests (Limitations)

- Can be quite complicated to setup and run, especially for distributed systems
- Can be very slow, making the feedback loop quite long
- May be very difficult or even impossible to run on a developer's machine
- Don't scale well compared to other kinds of automated tests
- Are unhelpful in pinpointing the hiding places of bugs found in production
- Their coarse-grained nature makes them unsuitable for checking that a single component, class, function, or method has been implemented correctly.

System Tests (Limitations)

Can be very difficult to use with distributed teams (especially in large projects)

Conclusions

- Having well chosen tests allows us to create quality systems
- Testability must be designed in the system
- Test First at all levels help
- Unit tests are not waste!

Books