Principles of Software Construction: Objects, Design, and Concurrency

Designing for Robustness in Large & Distributed Systems

Claire Le Goues

Vincent Hellendoorn

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Administrative

- Homework 6 has started
  - If you want the “discuss your design” bonus points, plan quickly! Thanksgiving is around the corner.
- Midterm grades out soon
  - Waiting for a few make-up exams
  - Will recap common mistakes hopefully on Thursday

Software Quality Assurance

What does quality mean in the context of modern Software Systems?

Software Quality Assurance

What does quality mean in the context of modern Software Systems? **It depends**, on user expectations. Some examples:

- Simplicity (of UI)
- Reliability
- Offering expected features
- Customizability
- Speed/Performance

Compare with design goals

Software Quality Assurance

How do you ensure quality in software systems?

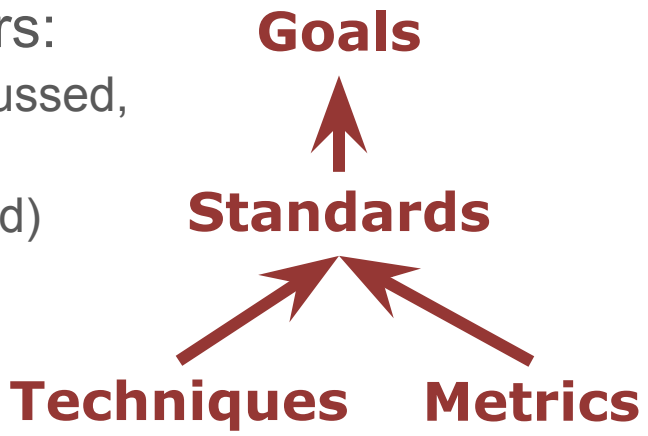
Software Quality Assurance

Is a well-established area with its own methods, models, and standards. It could fill a course of its own, but is so closely intertwined with software design that we teach some of it here.

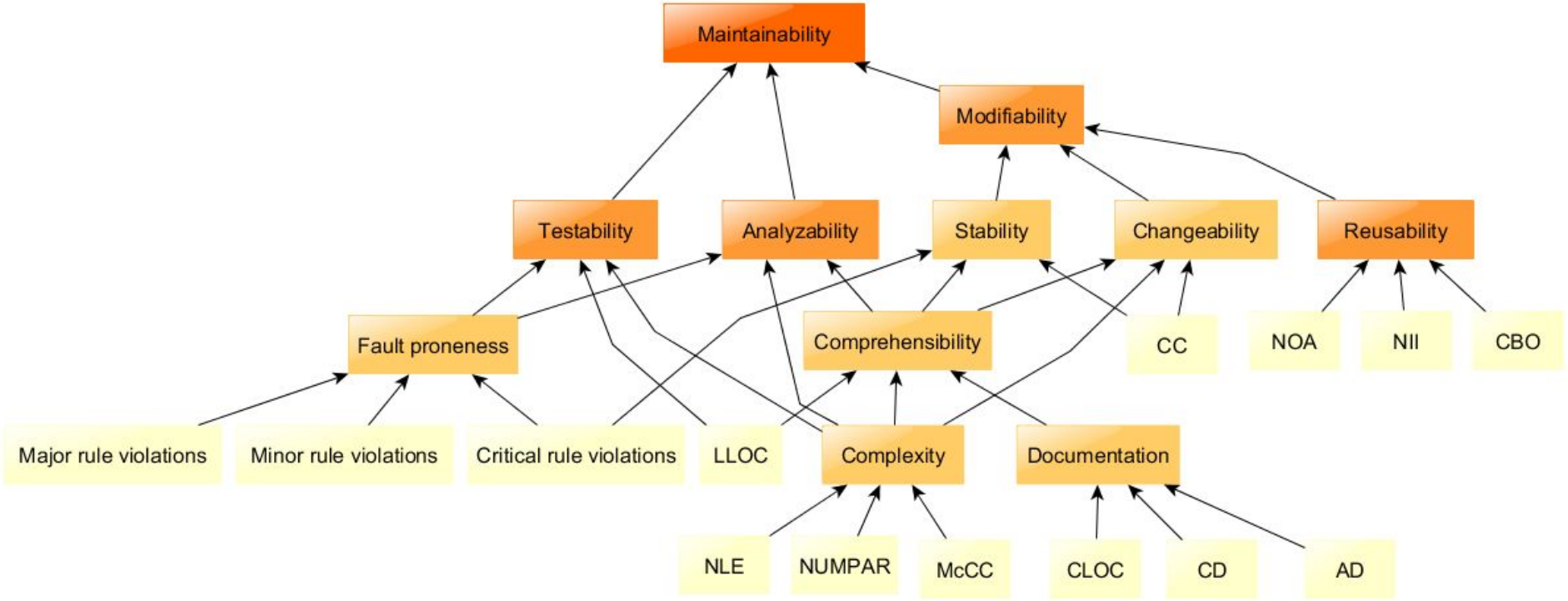
Software Quality Assurance

Is a well-established area with its own methods, models, and standards. It could fill a course of its own, but is so closely intertwined with software design that we teach some of it here.

- Much like design, can think of multiple tiers:
  - *Goals:* high-level objectives like the ones discussed, defined in the requirement specification
  - *Standards:* well-defined (incl. ISO-standardized) mappings of goals to measurable objectives
  - *Techniques* & *metrics:* tools & measurements used to ensure the system meets the standards



Excerpt of The ISO/EIC 9216 SQA Standard



https://en.wikipedia.org/wiki/ISO/IEC_9126#Developments

Software Quality Assurance

Is a well-established area with its own methods, models, and standards. It could fill a course of its own, but is so closely intertwined with software design that we teach some of it here.

- Factors in at every stage of software development
  - Model-driven design to create high-quality specifications
  - Designing using established design principles & patterns
  - Testing to measure conformance to specifications during development
  - Issue trackers to handle quality issues post-release

Software Quality Assurance

Is a well-established area with its own methods, models, and standards. It could fill a course of its own, but is so closely intertwined with software design that we teach some of it here.

- Is supported by a host of processes & tools
  - Code review
  - Testing
  - Version control
  - Coding practices (linters, documentation requirements)
  - Configuration management
  - SQA Management Plans (variations of processes, compare agile)

Today

We will talk about SQA specifically in the context of large & distributed systems, focusing primarily on achieving robustness

- Recapping: robustness challenges in distributed systems
- Testing distributed systems
  - With a discussion on test doubles
- Further Guidelines for improving robustness

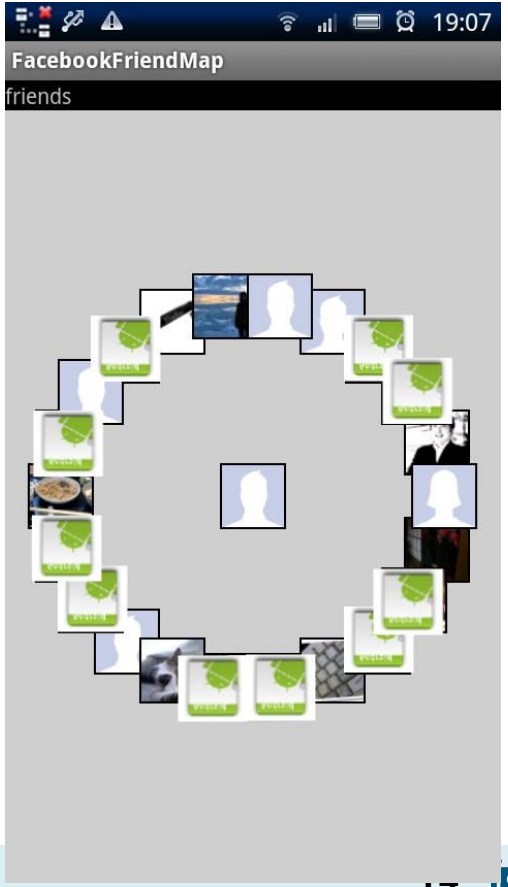
Where we are

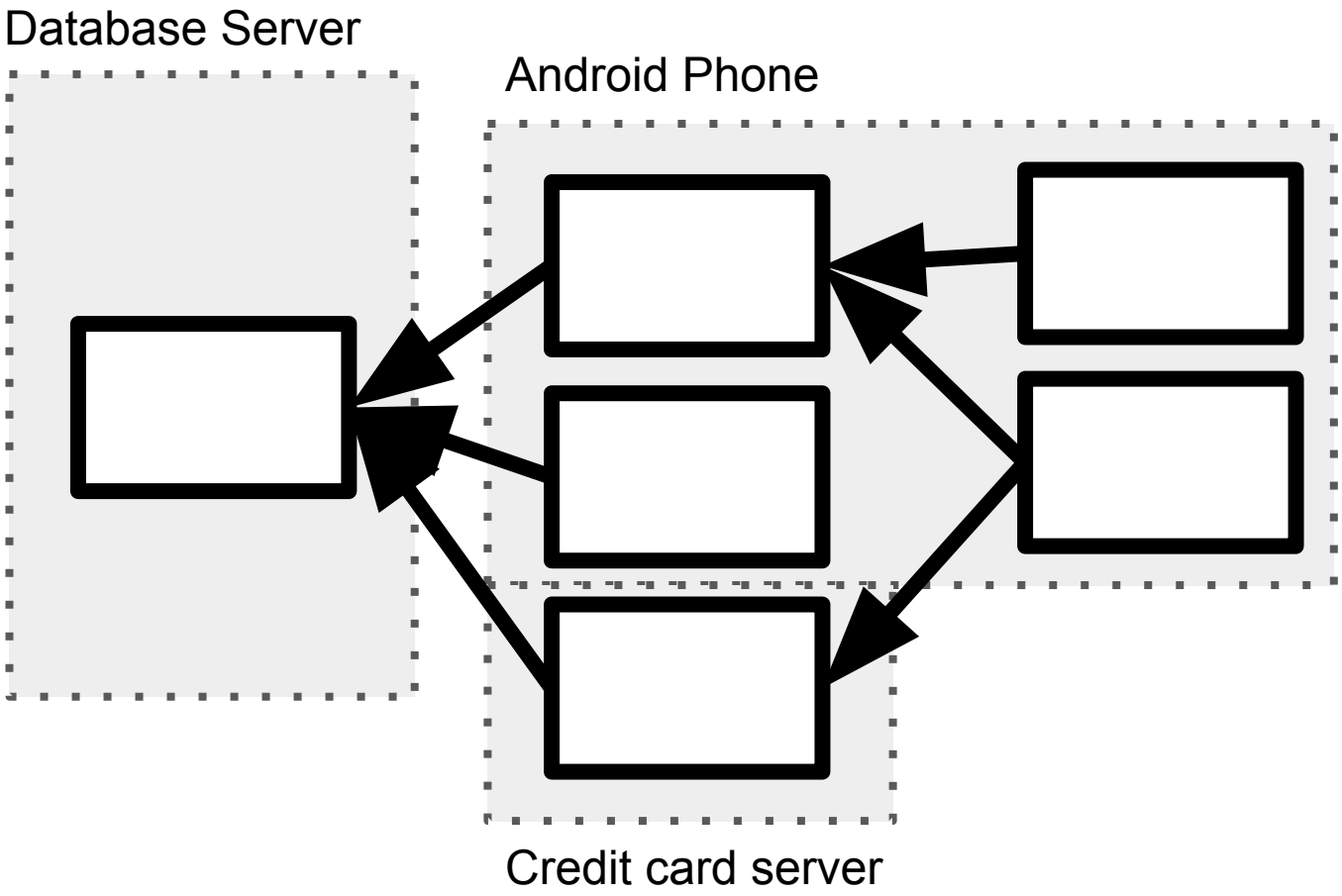
Design for	*Small scale:* One/few objects	*Mid scale:* Many objects	*Large scale:* Subsystems
understanding	Subtype Polymorphism ✓	Domain Analysis ✓	GUI vs Core ✓
change/ext.	Information Hiding, Contracts ✓	Inheritance & Del. ✓	Frameworks and Libraries ✓, APIs ✓
reuse	Immutability ✓	Responsibility Assignment, Design Patterns, Antipattern ✓	Module systems, microservices
robustness	Types	Promises/ Reactive P. ✓	**Designing for bustness**
...	Unit Testing ✓	Integration Testing ✓	CI ✓, DevOps, Teams

Recall: Modern software is dominated by systems composed of [components, APIs, modules], developed by completely different people, communicating over a network!

For example

- 3rd party Facebook apps
- Android user interface
- Backend uses Facebook data




Database Server
Android Phone
Credit card server

Testing (in) Distributed Systems

Testing in the Context of REST API Calls

Is conceptually no different:

- Test happy path
- Test error behavior

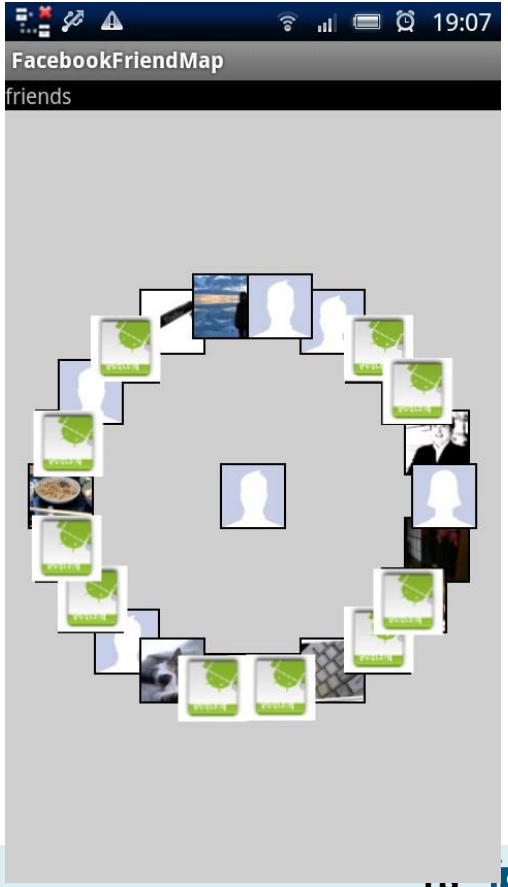
But different in instantiation:

- Correct timeout handling? Correct retry when connection down?
- Invalid response detected?
- Graceful degradation?

Need to understand possible error behavior first

Recall: Facebook Example

- 3rd party Facebook apps
- Android user interface
- Backend uses Facebook data



Assume an App


```
void buttonClicked() {
   render(getFriends());
}
List<Friend> getFriends() {
   Connection c = http.getConnection();
   FacebookAPI api = new FacebookAPI(c);
   return api.getFriends("john");
}
```

What Do We Test?


```
void buttonClicked() {
   render(getFriends());
}
List<Friend> getFriends() {
   Connection c = http.getConnection();
   FacebookAPI api = new FacebookAPI(c);
   return api.getFriends("john");
}
```

How Do We Test?


```
void buttonClicked() {
   render(getFriends());
}
List<Friend> getFriends() {
   Connection c = http.getConnection();
   FacebookAPI api = new FacebookAPI(c);
   return api.getFriends("john");
}
```

Eliminating the Android Dependency


```
@Test void testGetFriends() {
   assert getFriends() == ...;
}
List<Friend> getFriends() {
   Connection c = http.getConnection();
   FacebookAPI api = new FacebookAPI(c);
   return api.getFriends("john");
}
```

Eliminating the Remote Service Dependency?


```
@Test void testGetFriends() {
   assert getFriends() == ...;
}
List<Friend> getFriends() {
   Connection c = http.getConnection();
   FacebookAPI api = new FacebookAPI(c);
   return api.getFriends("john");
}
```

How about this call?

Recall: What will you do if

- Facebook withdraws its DNS routing information?
- **This affects testing too!**

https://blog.cloudflare.com/october-2021-facebook-outage/

Test Doubles

- Stand in for a real object under test
- Elements on which the unit testing depends (i.e. collaborators), but need to be approximated because they are
  - Unavailable
  - Expensive
  - Opaque
  - Non-deterministic
- Not just for distributed systems!

http://www.kickvick.com/celebrities-stunt-doubles

Eliminating the Remote Service Dependency


```
@Test void testGetFriends() {
   assert getFriends() == ...;
}
List<Friend> getFriends() {
   Connection c = http.getConnection();
   FacebookAPI api = new FacebookStub(c);
   return api.getFriends("john");
}
```

```
class FacebookStub
      implements FacebookAPI {
  void connect() {}
  List<Node> getFriends(String name)
    if (name.equals("john")) {
      return List.of(...);
    } // ...
  }
}
```

Types of Test Doubles

Fakes: Fully functional class with simplified implementation

Stubs: Artificial class that returns pre-configured data

Mocks: Instrumented variant of real class with fine-grained control

- Tend to be used interchangeably in practice
  - Most frameworks/libraries that support this focus on *mocking* (e.g., Mockito, ts-mocks), but also enable stubbing.
  - Rule of thumb: with stubs, you just assert against **values returned**, while with mocks, you assert against the **actual (instrumented) object**

Which Type Was This?


```
@Test void testGetFriends() {
   assert getFriends() == ...;
}
List<Friend> getFriends() {
   Connection c = http.getConnection();
   FacebookAPI api = new Facebook???(c);
   return api.getFriends("john");
}
```

```
class Facebook???
      implements FacebookAPI {
  void connect() {}
  List<Node> getFriends(String name)
    if (name.equals("john")) {
      return List.of(...);
    } // ...
  }
}
```

How About This?

```
10  public class InMemoryDatabase extends Database {
11
12      Map<String, Integer> accounts = new HashMap<>();
13
14      public void addAccount(String accountName, int password) {
15          this.accounts.put(accountName, password);
16      }
17
18      public int getPassword(String accountName) {
19          return this.accounts.get(accountName);
20      }
21  }
```

How Would You Test This?

```
@Test void testRecommendFriends() {
   ???;
}
List<Friend> recommendFriends(Person person) {
   Recommender m = AIFriendRecommender.newInstance();
   Map<Friend, Float> friendScores =
                          m.getRankedFriendCandidates(person);
   return friendScores.entrySet().stream()
                .sorted(e -> -e.getValue())
                .limit(10).map(e -> e.getKey())
                .collect(Collectors.toList());
}
```

Test Doubles

Concern that the third-party API might fail is not the only reason to use test doubles

- Most big, public APIs are extremely reliable
- Ideas for other reasons?

Test Doubles

Concern that the third-party API might fail is not the only reason to use test doubles

- Most big, public APIs are extremely reliable
- Ideas for other reasons?
  - Modularity/isolation: testing just our code speeds up development (conf. unit vs. integration testing), simplifies prototyping
  - Performance: APIs can be slow (network traffic, large databases, …)
    - Good test suites execute quickly; that pays off by enabling more test scenarios
  - Simulating other types of problems: changing APIs, slow responses, …

Fallacies of Distributed Computing by Peter Deutsch

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

How to Test Alternatives To:

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

Fault injection

- Mocks can emulate failures such as timeouts
- Allows you to verify the robustness of system

```
class FacebookSlowStub implements FacebookAPI {
  void connect() {}
  List<Node> getFriends(String name) {
    Thread.sleep(4000);
    if (name.equals("john")) {
      return List.of(...);
    } // ...
```

Fault injection


```
class FacebookErrorStub implements FacebookAPI {
  void connect() {}
  int counter = 0;
  List<Node> getFriends(String name) {
    counter++;
    if (counter % 3 == 0)
      throw new SocketException("Network is unreachable");
    else if (name.equals("john")) {
      return List.of(...);
    } //
```

How Test Doubles Help

1. Speed: simulate response without going through the API

```
class FakeFacebook implements FacebookInterface {
    void connect() {}
    List<Node> getFriends(String name) {
        if ("john".equals(name)) {
          List<Node> result=new List();
          result.add(…);
          return result;
        }
     }
}
```

How Test Doubles Help

1. Speed: simulate response without going through the API
2. Stability: guaranteed deterministic return, reduces flakiness

```
class FakeFacebook implements FacebookInterface {
    void connect() {}
    List<Node> getFriends(String name) {
        if ("john".equals(name)) {
          List<Node> result=new List();
          result.add(…);
          return result;
        }
     }
}
```

How Test Doubles Help

1. Speed: simulate response without going through the API
2. Stability: guaranteed deterministic return, reduces flakiness
3. Coverage: reliably simulate problems (e.g., return 404)

```
class FakeFacebook implements FacebookInterface {
    void connect() {}
    List<Node> getFriends(String name) {
        if ("john".equals(name)) {
          List<Node> result=new List();
          result.add(…);
          return result;
        }
     }
}
```

How Test Doubles Help

1. Speed: simulate response without going through the API
2. Stability: guaranteed deterministic return, reduces flakiness
3. Coverage: reliably simulate problems (e.g., return 404)
4. Insight: expose internal state

```
class FakeFacebook implements FacebookInterface {
    void connect() {}
    List<Node> getFriends(String name) {
        if ("john".equals(name)) {
          List<Node> result=new List();
          result.add(…);
          return result;
        }
     }
}
```

How Test Doubles Help

1. Speed: simulate response without going through the API
2. Stability: guaranteed deterministic return, reduces flakiness
3. Coverage: reliably simulate problems (e.g., return 404)
4. Insight: expose internal state
5. Development: presume functionality not yet implemented

```
class FakeFacebook implements FacebookInterface {
    void connect() {}
    List<Node> getFriends(String name) {
        if ("john".equals(name)) {
          List<Node> result=new List();
          result.add(…);
          return result;
        }
     }
}
```

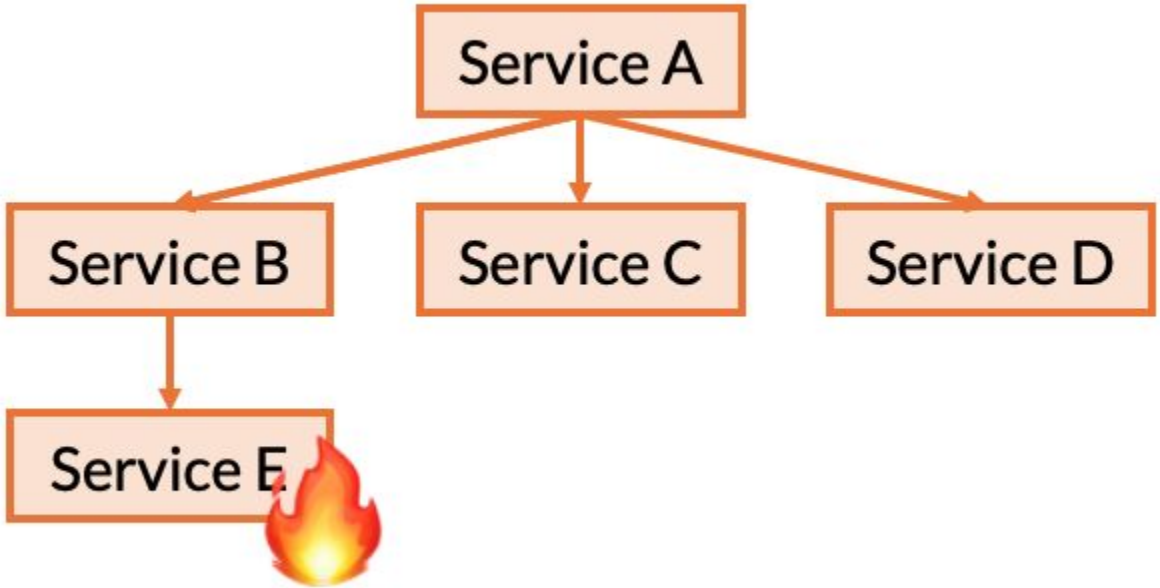
Design Implications

- Think about testability when writing code
- When a mock may be appropriate, design for it
- Hide subsystems behind an interface
- Use factories, not constructors to instantiate
- Use appropriate tools
  - Dependency injection or mocking frameworks

Chaos Engineering

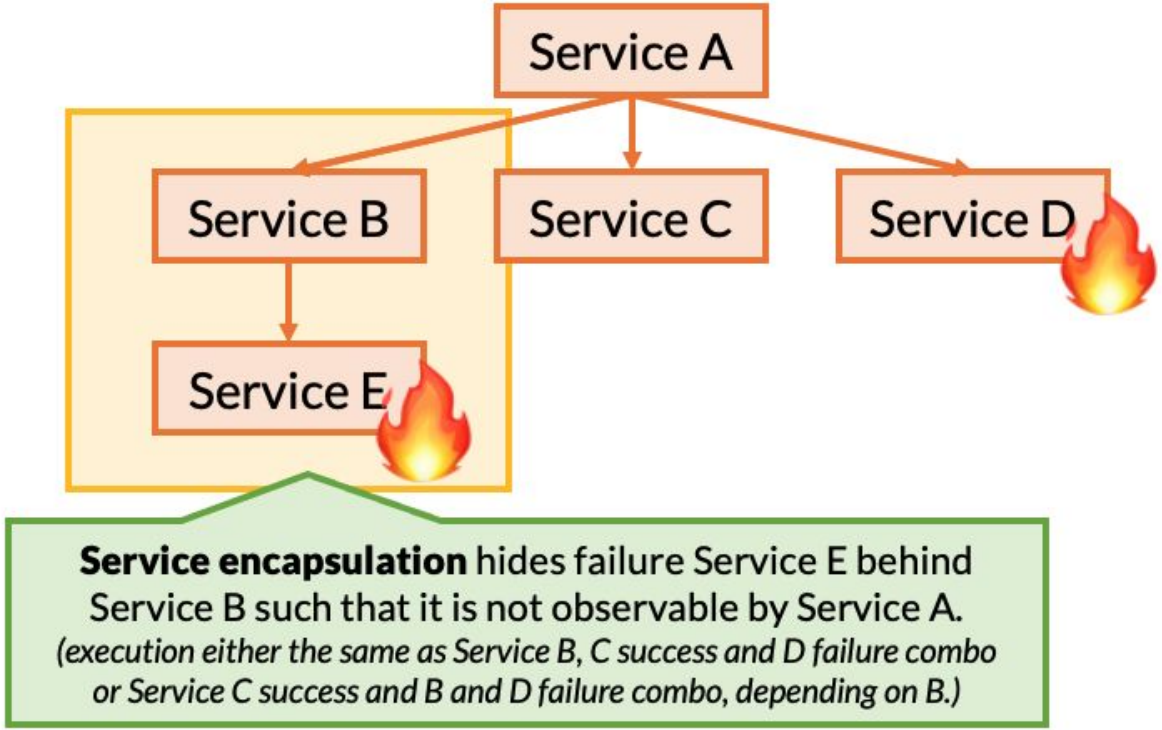
Experimenting on a distributed system in order to build confidence in the system's capability to withstand turbulent conditions in production

You Don’t Know It Works Until You Break It



http://christophermeiklejohn.com/filibuster/2021/10/14/filibuster-4.html

Handle Errors Locally



http://christophermeiklejohn.com/filibuster/2021/10/14/filibuster-4.html

Design: Testability

- Single responsibility principle
- Dependency Inversion Principle (DIP)
  - High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details. Details should depend upon abstractions.
- Law of Demeter: Don’t acquire dependencies through dependencies.
  - avoid: `this.getA().getB().doSomething()`
- Use factory pattern to instantiate new objects, rather than new.
- Use appropriate tools, e.g., dependency injection or mocking frameworks

Summary

- Software Quality plays into all aspects of software development
- Testing is a key quality control mechanism
- Distributed systems require rethinking testing
  - To achieve isolation, use test doubles
  - Which are useful for several reasons! Rapid prototyping, simulating failures, testing complicated behavior
- Robustness goes beyond test cases
  - To really error-proof a system, we have to stress-test it