

# Working Effectively with Legacy Code

Ch. 15, 16, 17

Kevin Yang

# My Application Is All API Calls

- Software systems don't live on their own, they usually need to talk to other bits of software
- Over time, we might still be able to see areas of code that don't touch an API, but they are embedded in a patchwork of unstable code.

## Why systems that are littered with library calls are harder to deal with ?

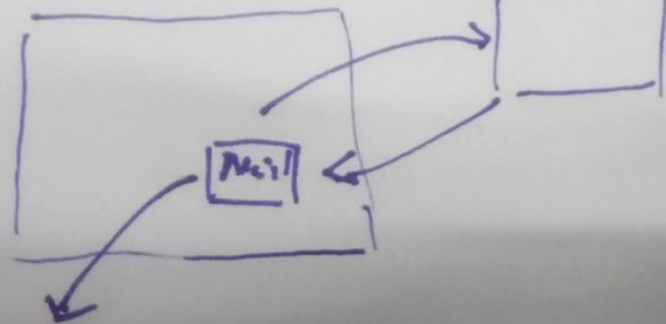
- All you can see are the API calls. Anything that would've been a hint at a design just isn't there.
- We don't own the API, we can't rename interfaces, classes, and methods to make things clearer for us, or add methods to classes to make them available to different parts of the code.

# Example

! MailSender.java

Mailing List Server

Folder

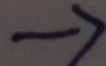


Mailbox

New Mail

Transport

Send



# Try to separate the code's responsibilities

1. We need something that can **receive** each incoming message and feed it into our system.
2. We need something that can just **send** out a mail message.
3. We need something that can make new messages for each incoming message, based on our roster of list recipients.
4. We need something that sleeps most of the time but wakes up periodically to see if there is more mail.

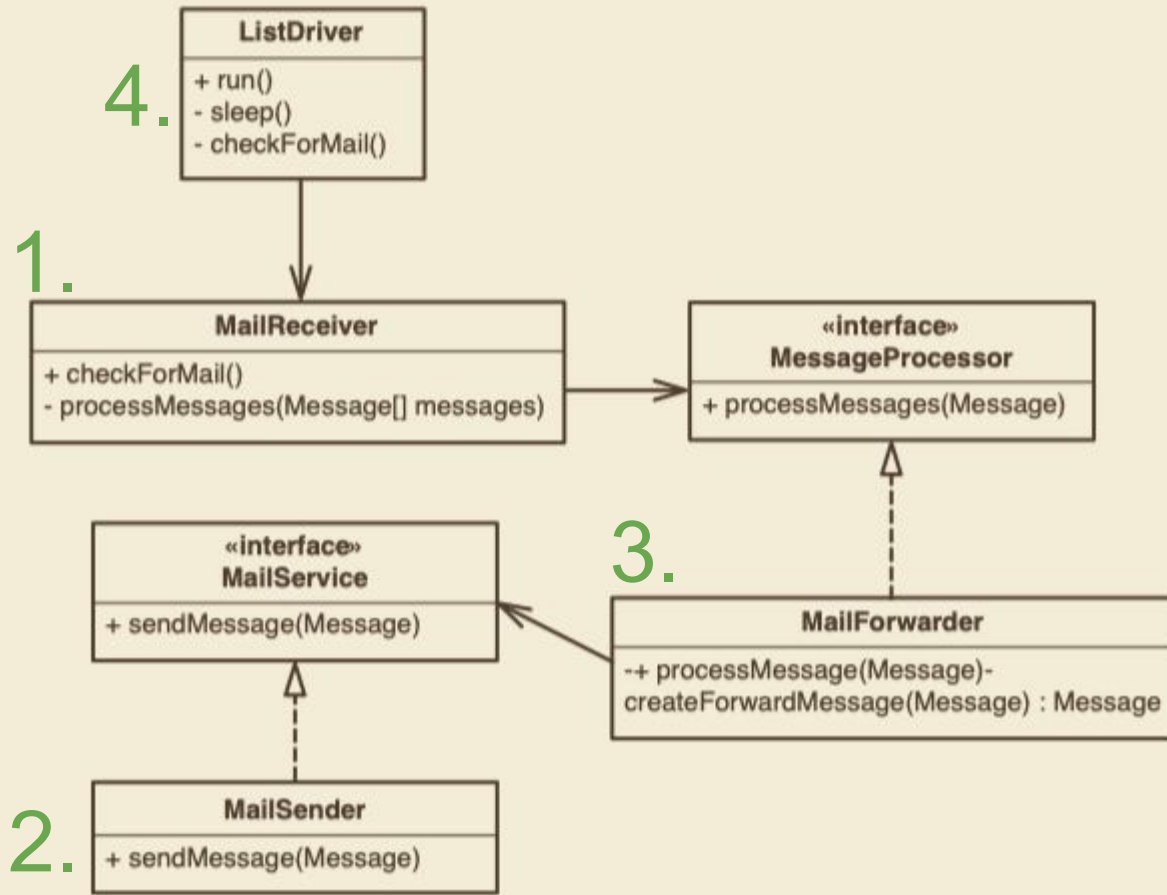


Figure 15.1 A better mailing list server.


- Skin and Wrap the API
  - Preserve Signatures
- Responsibility Based Extraction



## MailingListServer\_Legacy.java

```
1  ...
2
3  Session smtpSession = Session.getDefaultInstance (props, null);
4  Transport transport = smtpSession.getTransport ("smtp");
5  transport.connect (host.smtpHost, host.smtpUser, host.smtpPassword);
6  transport.sendMessage (forward, roster.getAddresses ());
7
8  ...
```

Session is a final class



# Responsibility-Based Extraction Result

 MailSender.java

- Skin and Wrap the API

- The API is **relatively small**.
- You want to completely **separate out dependencies** on a third-party library.
- You don't have tests, and you can't write them because you can't test through the API.

With the method, we would have the chance to get all of our code under test except for a thin layer of delegation from the wrapper to the real API classes.

- Responsibility Based Extraction

- The API is more complicated.
- You have a tool that provides a **safe extract method support**, or you feel confident that you can do the extractions safely by hand.

# I Don't Understand the Code Well Enough to Change It

- Notes/Sketching
- Listing Markup
  - Separating Responsibilities
  - Understanding Method Structure
  - Extract Methods
  - Understand the Effects of a Change
- Scratch Refactoring
- Delete Unused Code

# Notes/Sketching

These sketches **don't have** to be full-blown UML diagrams or function call graphs using some special notation—although, if things get more confusing, you might want to get more formal or neater to organize your thoughts.

The paper is just a tool **to make conversation** go easier and help us remember the concepts we're discussing and learning.

Checkout Ch17.

# Listing Markup

It is particularly useful with very long methods.

- Print the code that you want to work with.
- Separating Responsibilities

If several things belong together, put a special symbol next to each of them so that you can identify them.

- Understanding Method Structure

Often indentation in long methods can make them impossible to read.

- Extract Methods

To break up a large method

- Understand the Effects of a Change



# Scratch Refactoring

- Check out the code from your version-control system.
- Forget about writing tests.
- Extract methods, move variables, refactor it whatever way you want to get a better understanding of it—just don't check it in again.

# Risk of Scratch Refactoring

1. We make some gross mistake when we refactor that leads us to think that the system is doing something that it isn't.
2. We could get so attached to the way that the code turns out that we start to think about it in those terms all the time.

If we are **too attached to the end point of a Scratch refactoring**, we'll miss out on those insights.

But finally, you feel reasonably confident with scratch refactoring

# Delete Unused Code

- Unused code may cause confusion.
- You can always look for the old version from version-control.

# My Application Has No Structure

Application might have started out with a well-thought-out architecture, but over the years, under schedule pressure, they can get to the point at which nobody really understands the complete structure.

People can work for years on a project and not have any idea where new features are intended to go; they just know the **hacks** that have been placed in the system recently.

When teams aren't aware of their architecture, it tends to degrade.

- The system can be so complex that it takes a long time to get the big picture.
- The system can be so complex that there is no big picture.
- The team is in a very reactive mode, dealing with emergency after emergency so much that they lose sight of the big picture.

# The role of architect

- Architects are usually charged with the task of working out the big picture and making decisions that preserve the big picture for the team.
- An architect has to be out in the team, working with the members day to day, or else the code diverges from the big picture.
- Every person who is touching the code should know the architecture, and everyone else who touches the code should be able to benefit from what that person has learned.

# Telling the Story of the System

- Explain what the pieces of the design are and how they interact in few sentence.
- Next, pick the next most important things to say about the system.
- Keep going until you've said just about everything important about the core design of the system.



# Benefit from telling story

# Using JUnit as an example

JUnit has two primary classes. The first is called **Test**, and the other is called **TestResult**.

Users create tests and run them, passing them a **TestResult**.

When a test fails, it tells the **TestResult** about it. People can then ask the **TestResult** for all of the failures that have occurred.

# What simplifications do we have?

1. There are many other classes in JUnit.
2. Users don't create test objects. Test objects are created from test case classes via reflection.
3. Test isn't a class; it's an **interface**. The tests that run in JUnit are usually written in subclasses of a class named TestCase, which implements Test.
4. People generally don't ask TestResults for failures. TestResults register **listeners**, which are notified whenever a TestResult receives information from a test.

5. Tests report more than failures: They report the number of tests run and the number of errors.

Errors are problems that occur in the test that aren't explicitly checked for.

Failures are failed checks.

# Is that all?

Tests can be grouped into objects called suites. We can run a suite with a test result just like a single test. All of the tests inside it run and tell the test result when they fail.

# What simplifications do we have here?

1. TestSuites do more than just hold and run a set of tests. They also create instances of TestCase-derived classes via reflection.
2. Tests pass themselves to the TestResult class, which, in turn, calls the test-execution method back on the test itself. This back and forth happens at a rather **low level**. Thinking about it the simple way is kind of convenient and it's actually the way JUnit used to be in the early version.

# For Further Details



Check the books please.

💡 The more closer to the implementation phase, the more details we need.



# Naked CRC (Class, Responsibility, Collaborations)

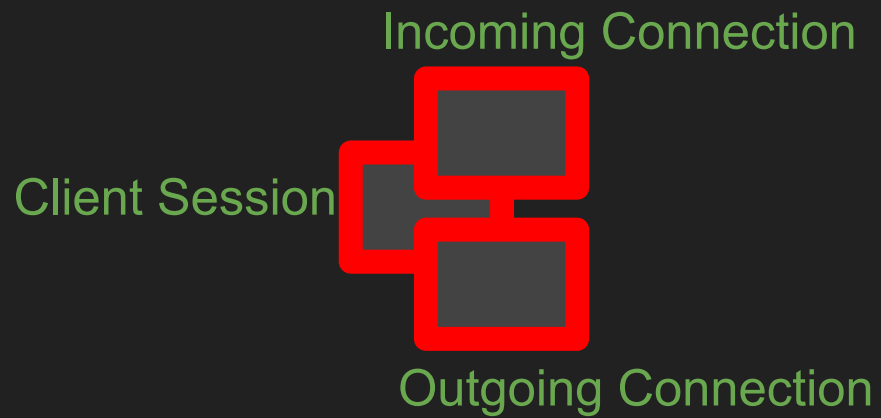
## Guidelines in Naked CRC

1. Cards represent instances, not classes.
2. Overlap cards to show a collection of them.

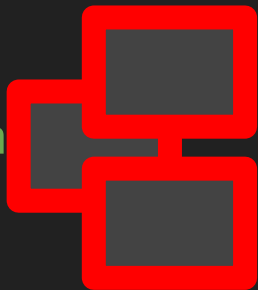
# Take some examples

Client Session





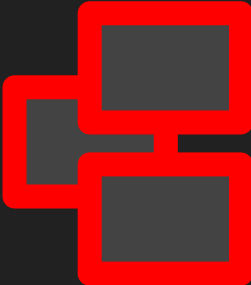
Client Session



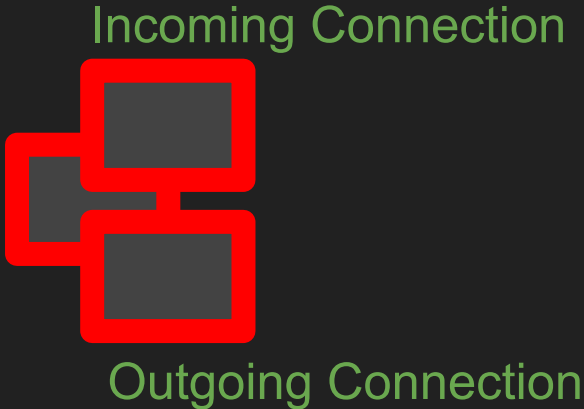
Server Session



Client Session



Server Session



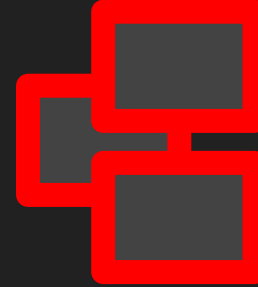
Vote manager



Client Session



Server Session



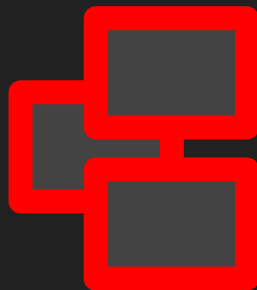
Vote manager



Many Server Session



Client Session



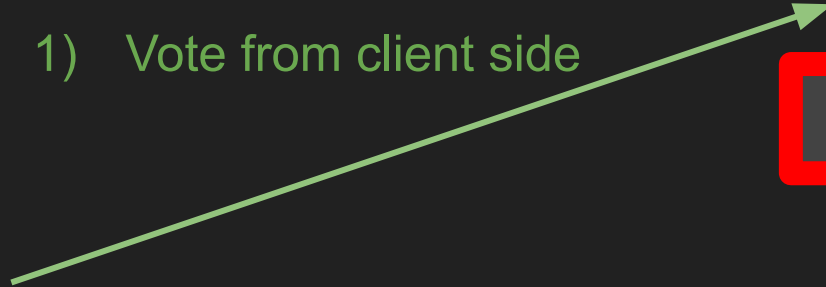
Vote manager



Server Session



1) Vote from client side



Client Session

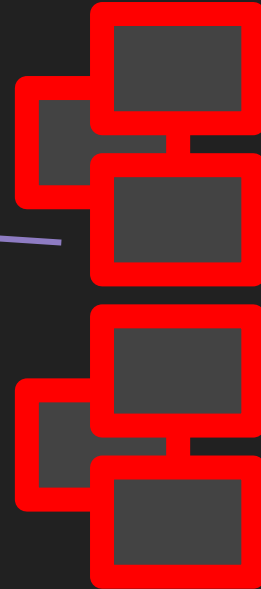




Vote manager



Server Session



Client Session



2) Response from  
server

Vote manager

3) Records the vote

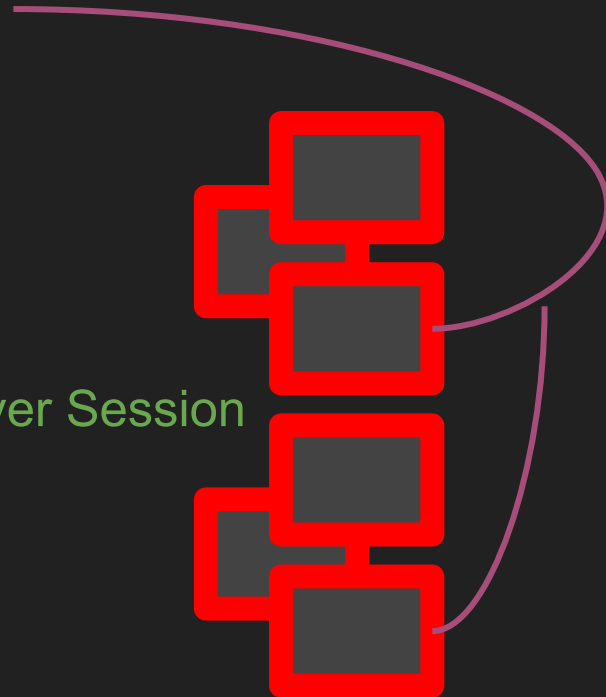
Server Session

Client Session



Vote manager

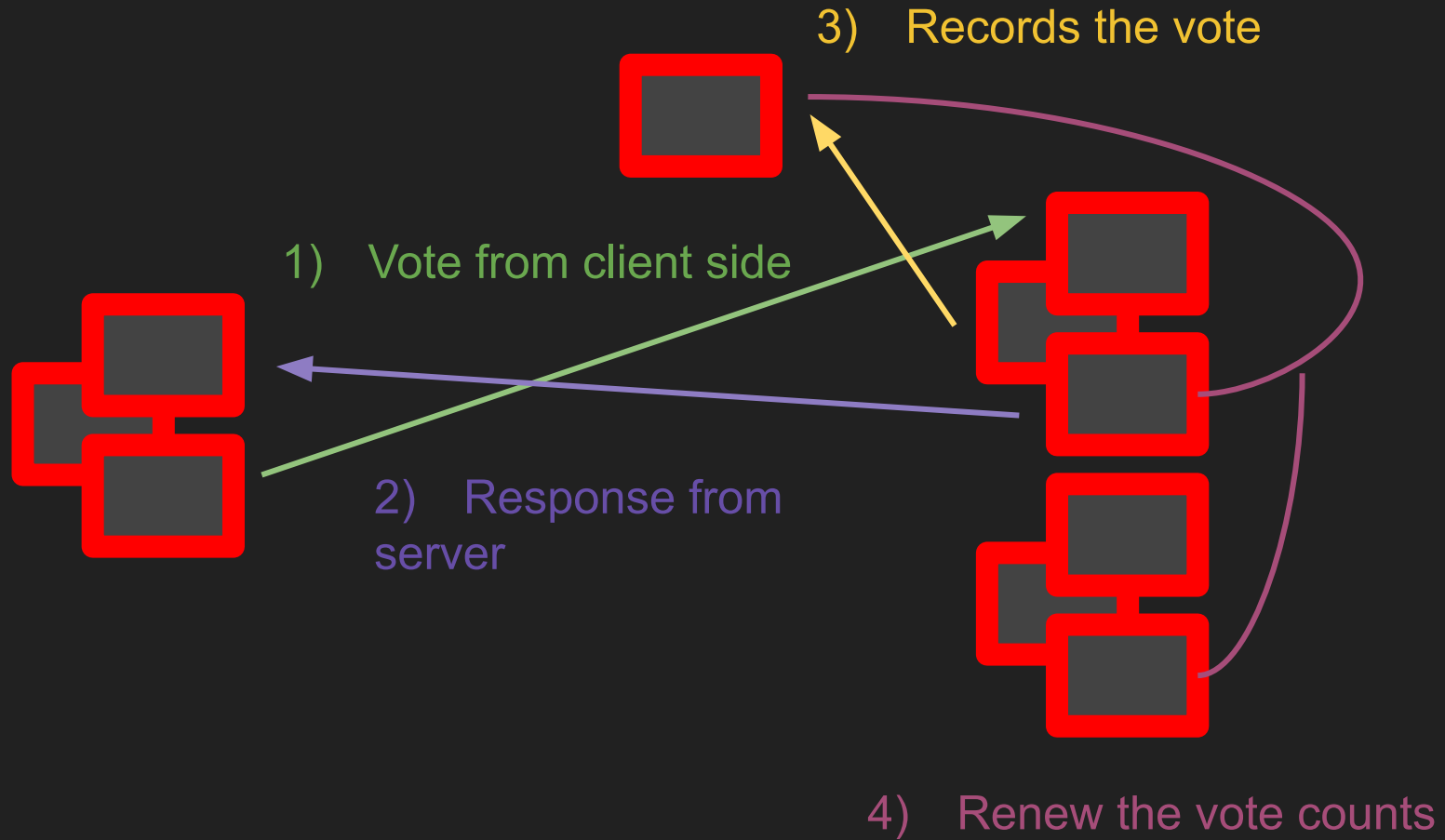
4) Renew the vote counts



Client Session



Server Session



# Conversation Scrutiny

In legacy code, it's tempting to avoid creating abstractions.

We are so distracted when we're trying to figure out these things, often we miss things that can give us additional ideas.

There is something mesmerizing about large chunks of procedural code: They seem to beg for more code

- Listen to conversations about your design.
- Are the concepts you're using in conversation the same as the concepts in the code?

# Conclusion

Design is design, regardless of when it happens in the development cycle.

One of the worst mistakes a team can make is it to feel that design is **over** at some point in development.

If design is “over” and people are still making changes, chances are good that new code will appear in poor places, and classes will bloat because **no one feels comfortable introducing new abstraction.**

Thank you for your attention