CS130: Software Engineering

Lecture 5: Testing
Refactoring

https://forms.gle/qiW4T64zWfcHMH7n7

A word: How’re ya doin’ today?
A tweet: Describe a bug you had
And what it took to fix

Assignment 2

Assignment 2

Featured:

- C++ / Boost
- Networking
- Docker
- Google Cloud Build
- Google Compute Engine

https://bit.ly/376jhg1

- What went well?
- What did not go well?
- What could be improved?

www.cs130.org down: Postmortem

Where's the problem?

```
steps:
- name: 'gcr.io/cloud-builders/gcloud-slim'
  entrypoint: 'bash'
  args:
  - '-c'
  - |
    mkdir _site
- name: 'jekyll/jekyll:3.8'
  args: [ 'jekyll', 'build' ]
  env:
  - 'JEKYLL_VERSION=3.8'
- name: '18fgsa/html-proofer:latest'
  args: [ '_site/', '--disable-external' ]
- name: 'gcr.io/cloud-builders/gcloud-slim'
  args:
  - 'compute'
  - 'scp'
  - '_site/'
  - 'chronos@${_GCE_INSTANCE}:${_DST_PATH}.tmp'
  - '--recurse'
  - '--zone=${_GCE_ZONE}'
  - '--ssh-key-expire-after=1m'
```

```
- name: 'gcr.io/cloud-builders/gcloud-slim'
  args:
  - 'compute'
  - 'ssh'
  - 'chronos@${_GCE_INSTANCE}'
  - '--zone=${_GCE_ZONE}'
  - '--ssh-key-expire-after=1m'
  - '--command'
  - |
    if [ -d ${_DST_PATH}.last ]; then
      rm -rf ${_DST_PATH}.last;
    fi;
    if [ -d ${_DST_PATH} ]; then
      mv ${_DST_PATH} ${_DST_PATH}.last;
    fi;
    mv ${_DST_PATH}.tmp ${_DST_PATH}
```

What's the problem?

In one step:

- scp _site/ X.tmp

In subsequent step:

- If X.last exists, remove it
- If X exists, mv it to X.last
- Move X.tmp to X

How can we solve it?

```
- 'compute'
- 'scp'
- '_site/'
- 'chronos@${_GCE_INSTANCE}:${_DST_PATH}.tmp'
```

… then …

```
if [ -d ${_DST_PATH}.last ]; then
  rm -rf ${_DST_PATH}.last;
fi;
if [ -d ${_DST_PATH} ]; then
  mv ${_DST_PATH} ${_DST_PATH}.last;
fi;
mv ${_DST_PATH}.tmp ${_DST_PATH}
```

The fix is in

scp _site/ X.tmp

scp _site/ X.tmp

rm X.last

X -> X.last

X.tmp -> X

rm X.last

X -> X.last

X.tmp -> X (error: X.tmp not found)

scp _site/ X.1

scp _site/ X.2

rm X.last

X -> X.last

X.1 -> X

rm X.last

X -> X.last

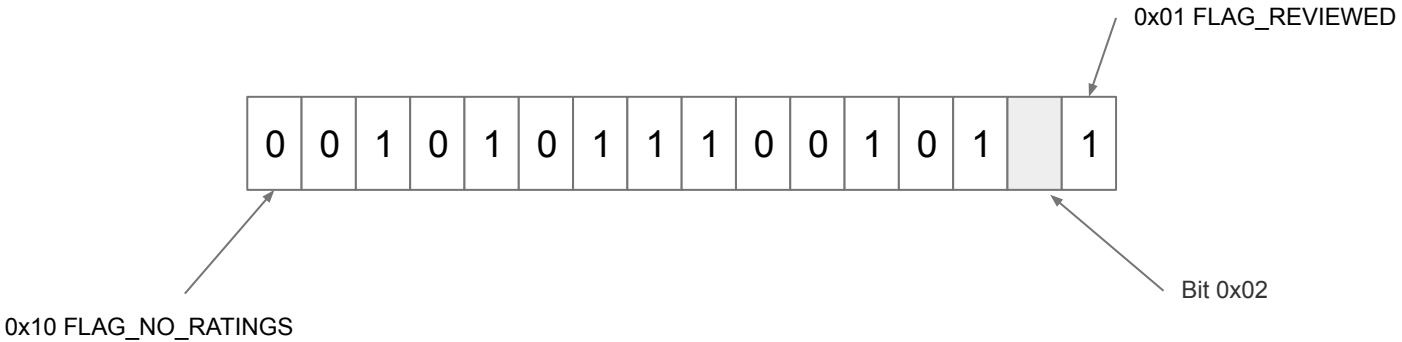
X.2 -> X

A Note About Merging

Know what you are merging!

```
+<<<<<<< .working
      FLAG_XXXXXXXXXXXXXXXX = 0x0002000000000000 # comment about what the flag does
+=======
+      FLAG_YYYYYYYY          = 0x2000000000000 # comment about what the flag does
+
+>>>>>>> .merge-right.r85508
```

These both point to the same bit

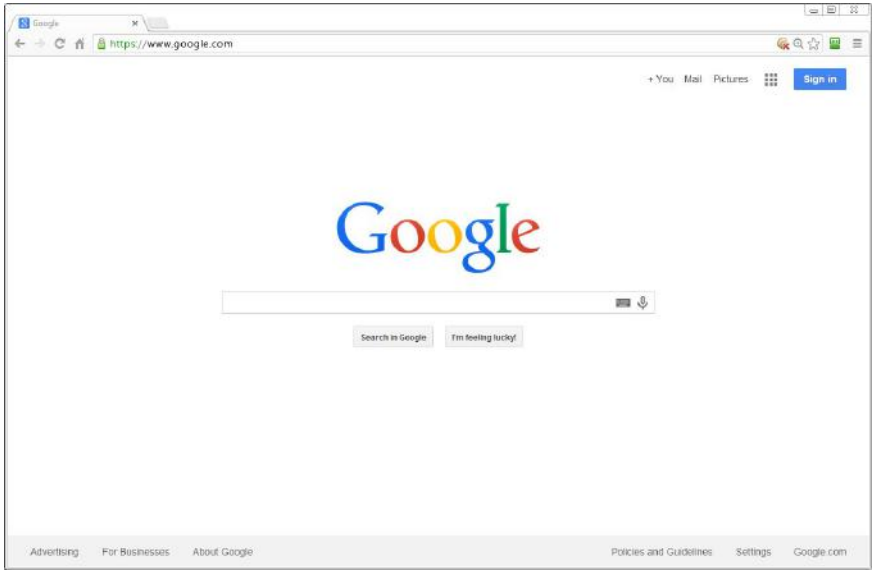


Know what you are merging!

```
+<<<<<<< .working
      FLAG_REJECTED_CHILD_PORN_CONTENT = 0x0002000000000000 # video has been confirmed in video review as child porn content
+=======
+       FLAG_IS_PROMOTED              = 0x2000000000000 # video is being promoted with PYV
+
+>>>>>>> .merge-right.r85508
```

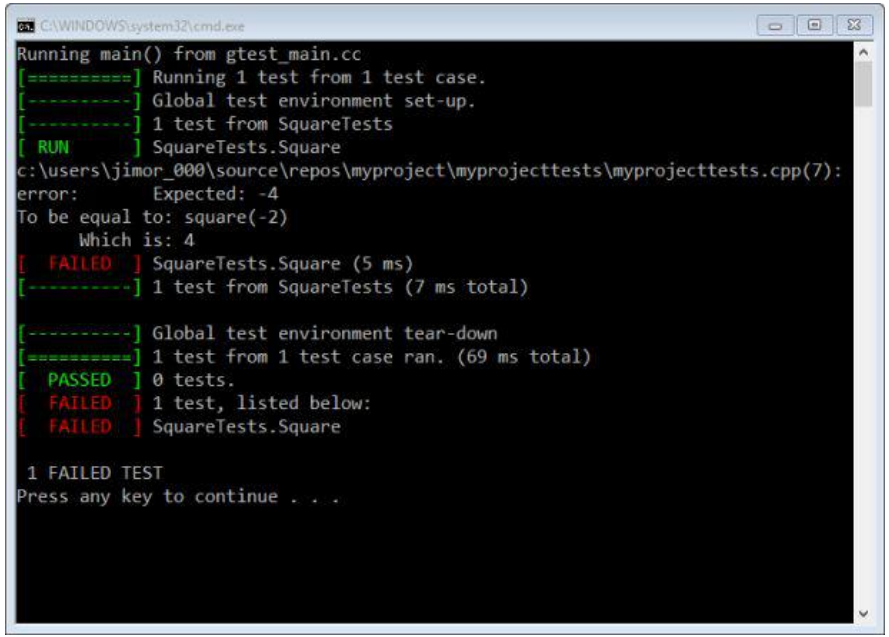
Testing

Testing a web server



- Just enter your URL
- See if it returns something...

But, you want it to be repeatable



As we've discussed earlier:

- You want one command to run all the tests against your web server.
- You want the test to be hermetic; i.e. not using a web browser if possible.
- For unit tests, you want to test only your components, not boost or the kernel.
- For integration tests, you need to invent a way to act like a simple web browser.

Author : I tested it and it works

99836	author	current_office = validation_base.FormField(validators.ThreeLetterValidator, _("I call shenanigans"))
99836	author	future_office = validation_base.FormField(validators.ThreeLetterValidator, _("I call shenanigans"))
99836	author	party_affiliation = validation_base.FormField(validators.ThreeLetterValidator, _("I call shenanigans"))
99836	author	genre = validation_base.FormField(validators.TextValidator, _("I call shenanigans"), max_length=3)
99836	author	formation_date = validation_base.FormField(validators.DateValidator, _("I call shenanigans"))
99836	author	record_label = validation_base.FormField(validators.TextValidator, _("I call shenanigans"), max_length=128)
99836	author	label_type = validation_base.FormField(validators.TextValidator, _("I call shenanigans"), max_length=3)
99836	author	members = validation_base.FormField(validators.TextValidator, _("I call shenanigans"), max_length=500)
99836	author	influences = validation_base.FormField(validators.TextValidator, _("I call shenanigans"), max_length=500)

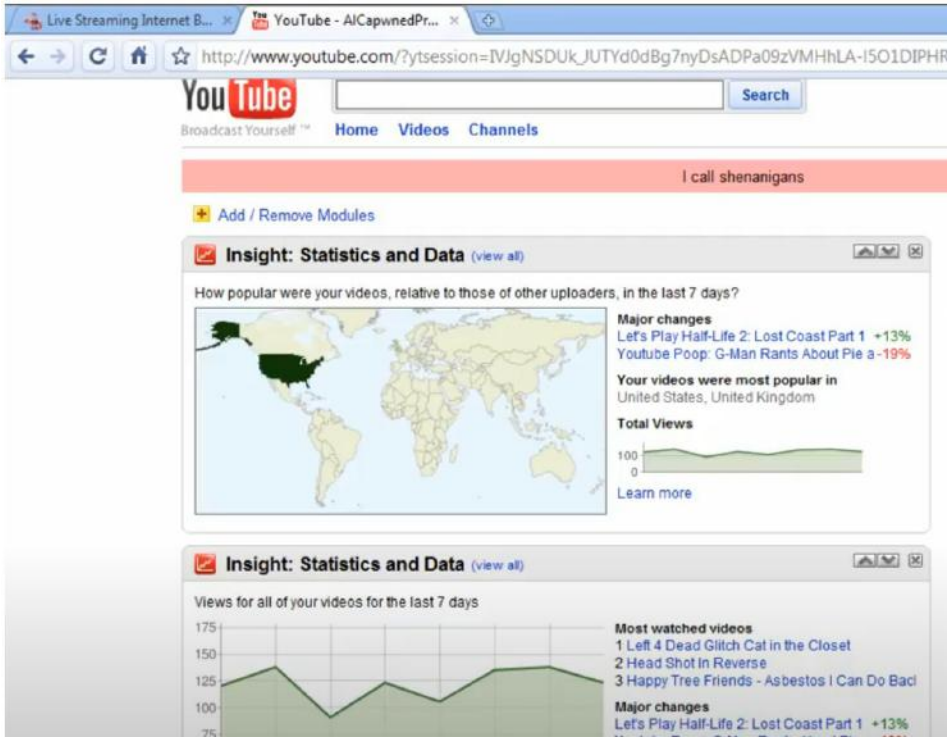
Which error caused "I call shenanigans???"

Solution!

99836	author	current_office = validation_base.FormField(validators.ThreeLetterValidator, _("I call shenanigans1"))
99836	author	future_office = validation_base.FormField(validators.ThreeLetterValidator, _("I call shenanigans2"))
99836	author	party_affiliation = validation_base.FormField(validators.ThreeLetterValidator, _("I call shenanigans3"))
99836	author	genre = validation_base.FormField(validators.TextValidator, _("I call shenanigans4"), max_length=3)
99836	author	formation_date = validation_base.FormField(validators.DateValidator, _("I call shenanigans5"))
99836	author	record_label = validation_base.FormField(validators.TextValidator, _("I call shenanigans6"), max_length=128)
99836	author	label_type = validation_base.FormField(validators.TextValidator, _("I call shenanigans7"), max_length=3)
99836	author	members = validation_base.FormField(validators.TextValidator, _("I call shenanigans8"), max_length=500)
99836	author	influences = validation_base.FormField(validators.TextValidator, _("I call shenanigans9"), max_length=500)

This was accidentally submitted :0

Users noticed.



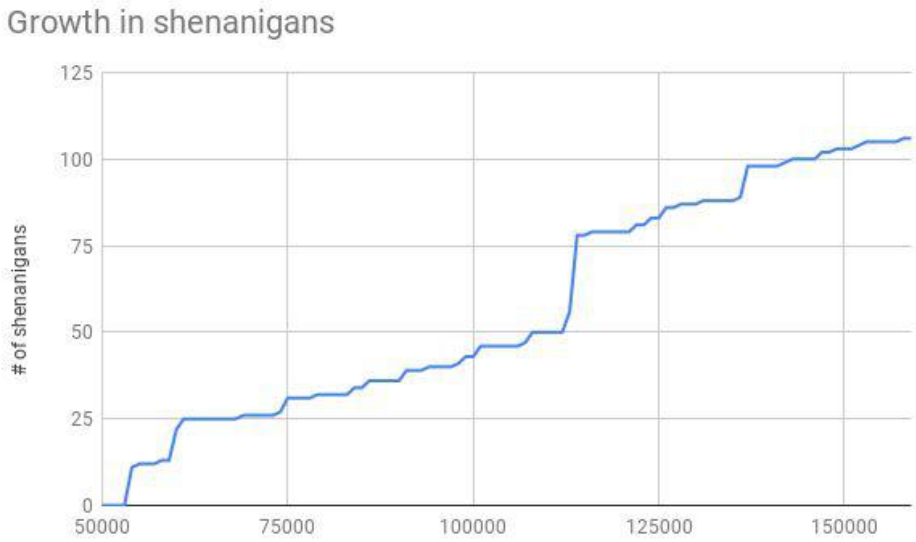
Actual bug report :

Steps to reproduce:

1. Sign into account
2. Try and edit channel description
3. In the "Influences" section, a red bar at the top appears saying "I call shenanigans9"

Seems this is occurring to various users as **users have started going to the channel of shenanigans9 (poor guy) and complaining there** that they're also experiencing the issue. Please see this forum thread for more info.

Bad practices propagate



By the time we caught it, there were over 100 error conditions returning “I call shenanigans.”

Be sure to test the right things

```
int main(int argc, char* argv[]) {
    boost::asio::io_service io_service;
    tcp::acceptor a(*io_service, tcp::endpoint(
        tcp::v4(), 12345));
    while (true) {
        tcp::socket sock(*io_service);
        a.accept(sock);

        auto error = ProcessRequest(&sock);
        if (error) {
            printf("ProcessRequest() failed: %d: %s\n",
                   error.value(), error.message().c_str());
        }
    }
    return 0;
}
```

- You probably have something that looks like this.
- You could test this boilerplate, but no real need to do so in a unit test:
  - Doesn't change very often
  - Small amount, trivial code
  - Obvious when broken
  - Annoying to write such tests
- You should test this via an integration test

Testing boundaries

```
int main(int argc, char* argv[]) {
    boost::asio::io_service io_service;
    tcp::acceptor a(*io_service, tcp::endpoint(
        tcp::v4(), 12345));
    while (true) {
        tcp::socket sock(*io_service);
        a.accept(sock);

        auto error = ProcessRequest(&sock);
        if (error) {
            printf("ProcessRequest() failed: %d: %s\n",
                   error.value(), error.message().c_str());
        }
    }
    return 0;
}
```

- Note the boundary here where the socket is handed off to another function.
- This code was intentionally factored in such a way to separate connection processing from socket processing.
- Gives us a place to create a test!

Testing over the network?

```
boost::system::error_code ProcessRequest(tcp::socket* sock) {
    [...]
}

TEST_F(ProcessRequestTest, ReadRequest) {
    Socket socket;

    // false == no error
    EXPECT_FALSE(ProcessRequest(&socket));
}


TEST_F(ProcessRequestTest, ReadRequestFailsGracefully) {

    // This socket needs to be created... to fail?
    Socket socket;

    // false == no error
    EXPECT_FALSE(ProcessRequest(&socket));
}
```

- In future lectures, we'll look at mocks and other ways to make this test more isolated
- Definitely don't want to require a fully running webserver to run this test (that's an integration test!)
- We'd rather spend time testing what happens in `ProcessRequest()`, that is where our web server really lives.

Factoring Out a Handler

```
void ProcessRequest(tcp::socket* sock) {
    while (true) {
        const int kBufferLength = 1024;
        char data[kBufferLength];

        boost::system::error_code err;
        const size_t length =
            sock->read_some(boost::asio::buffer(data), err);
        HandleRequest(data, length));
    }
}

void HandleRequest(const char* buf, const size_t length) {
    [...]
}
```

- The thing we actually want to test is `HandleRequest()`
- And conveniently, its signature is now `void(const char*, const size_t)`
- This is a much more easily testable signature.

Testing your actual functionality

```
void HandleRequest(const char* buf, const size_t length) {
    [...]
}

TEST_F(HandleRequestTest, Simple) {
    EXPECT_EQ(/* what do I write here? */);
}
```

- Now that `HandleRequest()` is factored out, you can test it directly.
- But, what can I expect? It only really has side effects.

More testable functionality

~~void HandleRequest(const char* buf, const size_t length) {~~

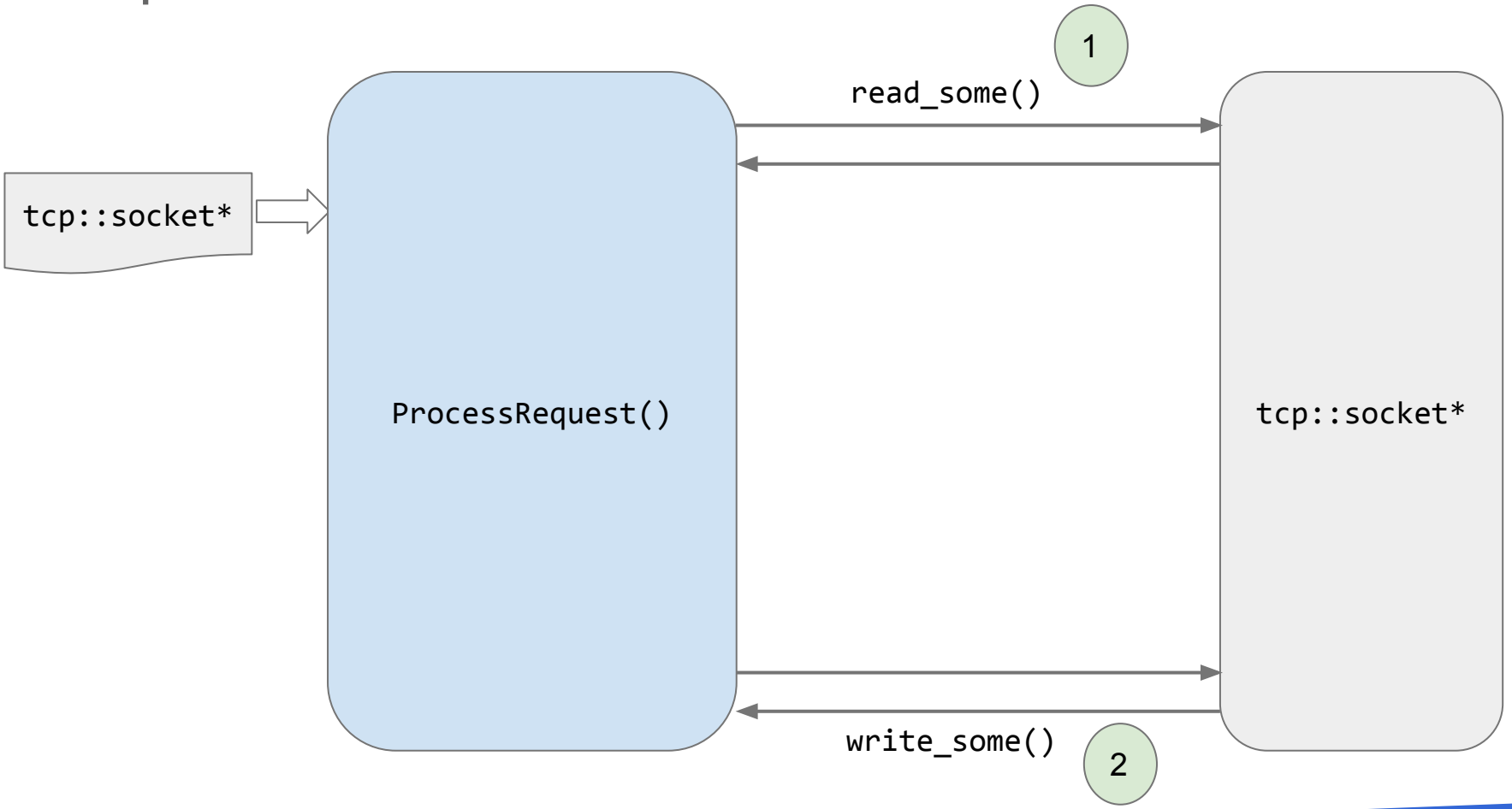
```
std::string HandleRequest(const char* buf,
                          const size_t length) {
    [...]
}

TEST_F(HandleRequestTest, Simple) {
    EXPECT_THAT(HandleRequest("My test input.", 14),
                HasSubstr("My expected output"));
}
```

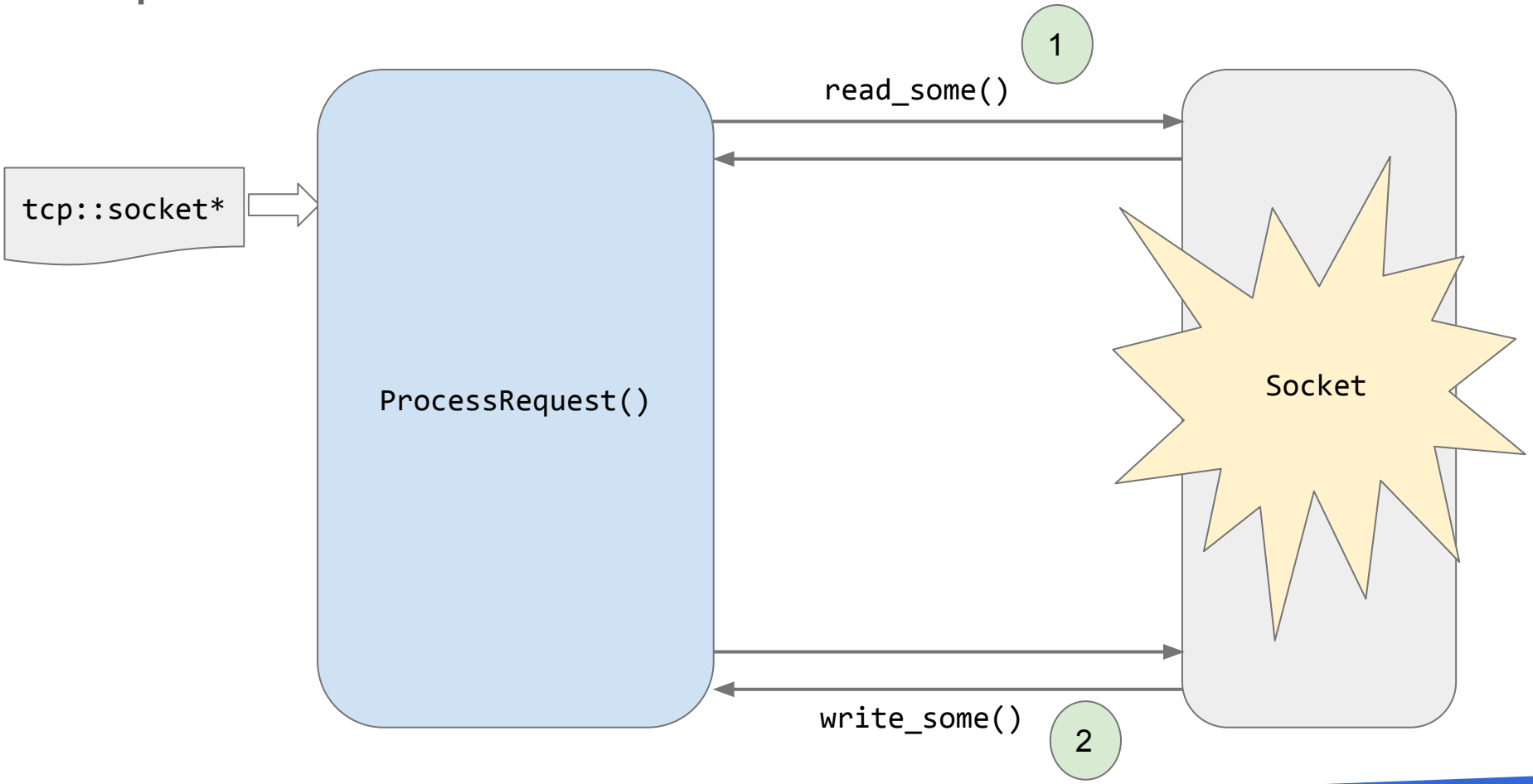
- We can expose the side-effects* by returning a string containing the response.
- Now we can directly inspect what happened.
- Bonus points for using a gmock matcher that makes the expectation more readable.

*The eagle-eyed among you may have noticed that we never wrote back to the socket in `ProcessRequest()`. We returned something that we'd eventually write. Is this the only option?

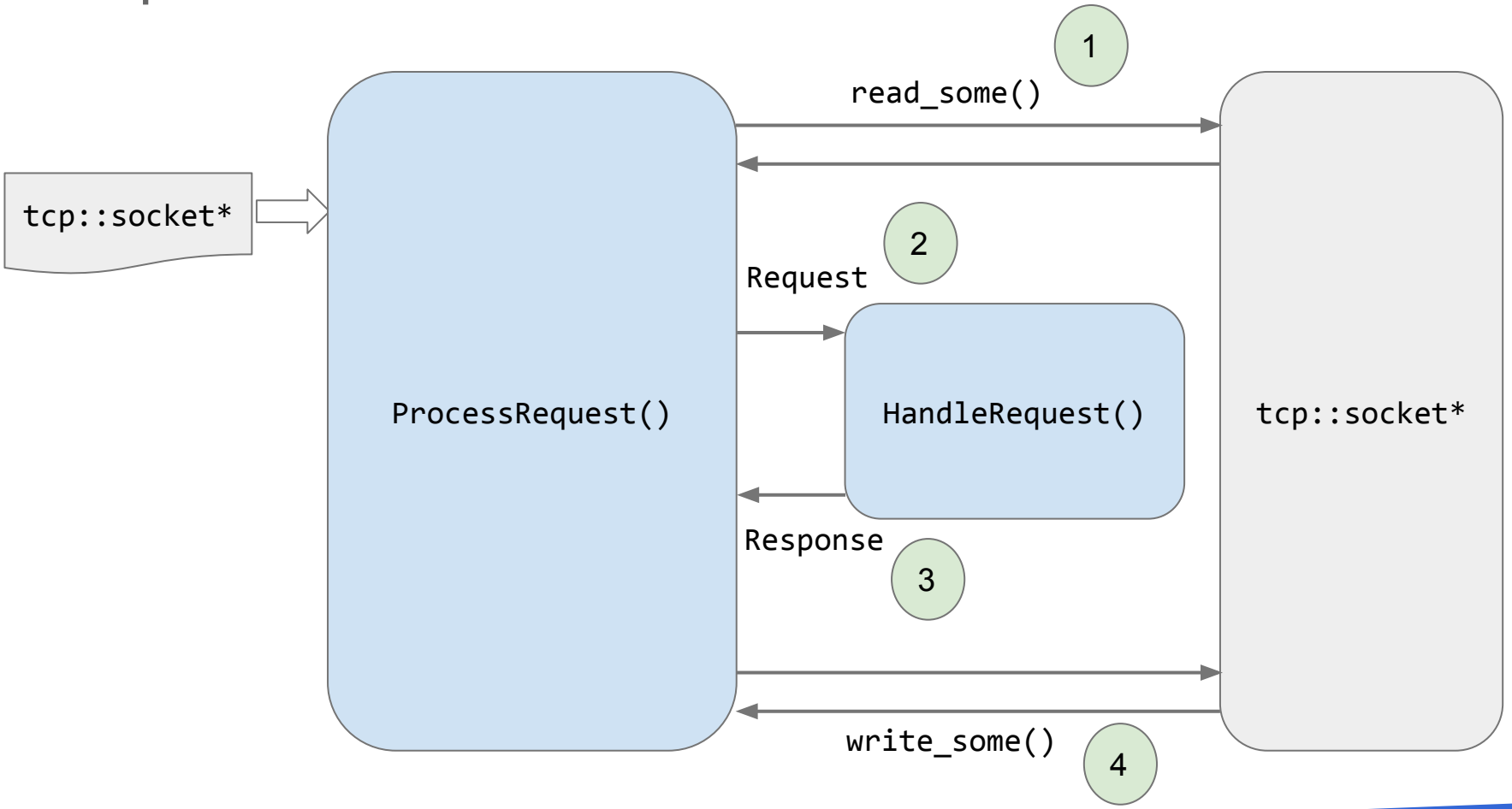
Recap: Code structure before refactor



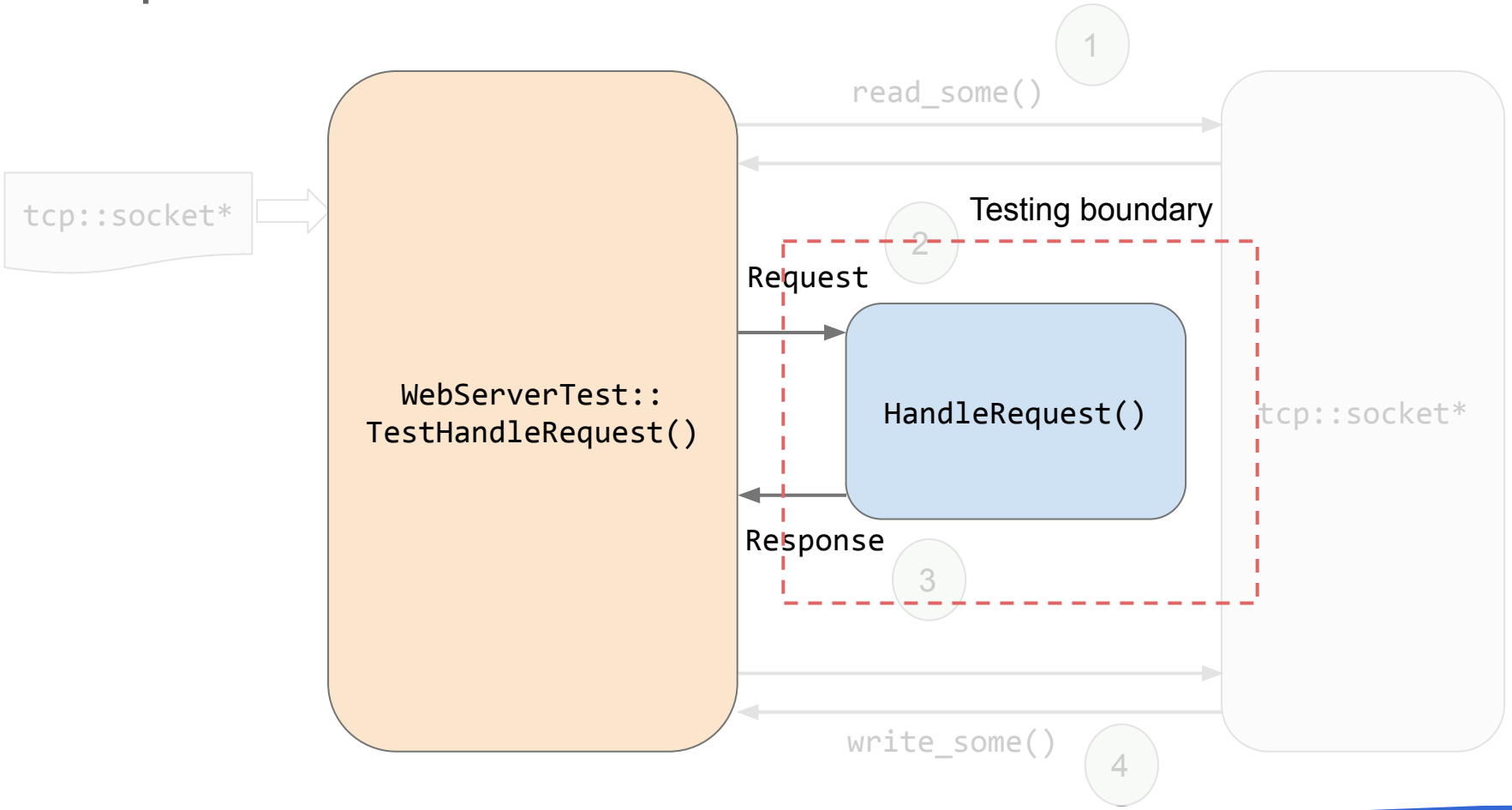
Recap: Code structure before refactor



Recap: Code structure after refactor



Recap: Code structure after refactor

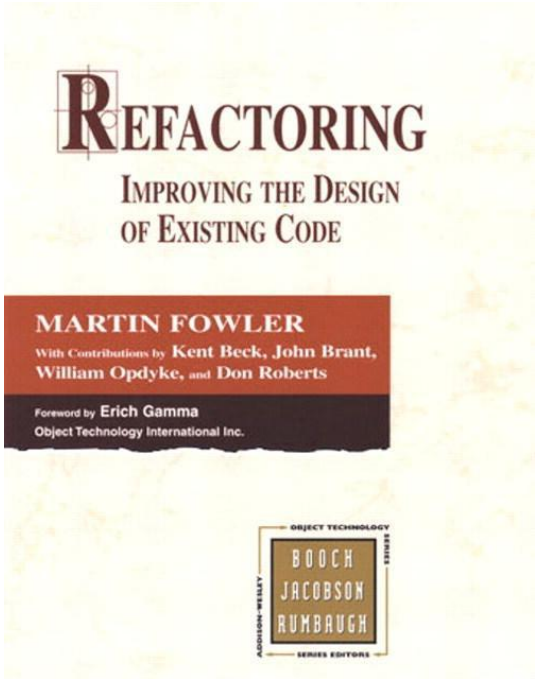


Refactoring

What is this “refactoring” you speak of?

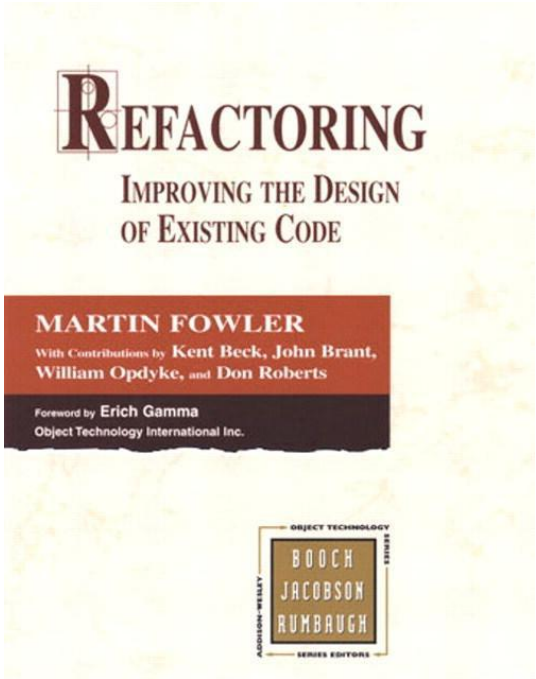
- Simply, it is rewriting (editing in the traditional sense) code to improve some property.
- In this case, we are restructuring the code to be more testable.
- Could also refactor to make it more maintainable:
  - Divide up long functions (extract method)
  - Make a class do fewer things (extract class)

Refactoring



- There is literally a book about it.
- Code is read more than it is written, so care must be taken to make it reader friendly.
- However, code gets more complex with time through natural evolution.
- Refactoring is all about moving code around to make it more maintainable without changing the behavior.

Refactoring



- Each change is small and incremental, so the changes are more likely to be safe.
- Best practice: create tests ahead of time so the tests can verify correct behavior after the changes.
- The book has a catalogue of refactorings and associated “smells” that suggest such a refactor should be done.

All code is pseudocode!

In the slides:

- Code is not syntactically correct
- Only the demonstrative parts of the code are shown

Refactoring and Mocks

- Mocks can be valuable when you are the client of large classes you don't want to test.
- Think back to `NginxConfigParser`, which was a nice clean example.
- However, mocking out the TCP socket didn't really help us, and made the test code much more complex.

- Costly because you have to change the code's structure materially, usually by adding interfaces.
- Code often ends up a bit cleaner (as this case), sometimes you have to poke ugly holes for testing.

Refactoring Examples

Before:

```
void PrintOwing() {
    PrintBanner();

    // print details
    printf("name: %s\n", name_);
    printf("amount: %d\n", GetOutstanding());
}
```

- Extract method when a function gets too long or has a useful internal boundary.
- We used this to extract `ProcessRequest()` and later `HandleRequest()`

Refactoring Examples

Before:

```
void PrintOwing() {
    PrintBanner();

    // print details
    printf("name: %s\n", name_);
    printf("amount: %d\n", GetOutstanding());
}
```

After:

```
void PrintOwing() {
    PrintBanner();
    PrintDetails();
}

void PrintDetails() {
    printf("name: %s\n", name_);
    printf("amount: %d\n", GetOutstanding());
}
```

- Extract method when a function gets too long or has a useful internal boundary.
- We used this to extract `ProcessRequest()` and later `HandleRequest()`

Refactoring Examples

Before:

```
void Init(NginxConfigParser* parser,
          MimeTypeMap* mime,
          HandlerMap* handlers,
          [... many more things ...]) {
    [...]
}
```

- Introduce param object when a method's param list gets too long
- Certain functions have many params
  - Often triggered by dependency injection or tunability
- Can create a param object (also known as an options struct) to encapsulate these params
- Nice side effect: you can define defaults and have a bit more control over values before the function executes

Refactoring Examples

Before:

```
void Init(NginxConfigParser* parser,
          MimeTypeMap* mime,
          HandlerMap* handlers,
          [... many more things ...]) {
    [...]
}
```

After:

```
struct HttpServerOptions {
    HttpServerOptions() { [... set defaults ...] }

    NginxConfigParser* parser = nullptr;
    MimeTypeMap* mime = nullptr;
    HandlerMap* handlers = nullptr;
    [... many more things ...]
}

void Init(const HttpServerOptions& options) {
    [...]
}
```

- Introduce param object when a method's param list gets too long
- Certain functions have many params
  - Often triggered by dependency injection or tunability
- Can create a param object (also known as an options struct) to encapsulate these params
- Nice side effect: you can define defaults and have a bit more control over values before the function executes

Refactoring Examples

Before:

```
if (time() - start > 86400) {
    [...]
}
```

- Replace magic number with symbolic constant.
- There is nothing worse than random numbers strewn around code that have no clear documentation (i.e., why'd they pick that number instead of another?!).
- Instead, use symbolic constants to make the code self-documenting.
- We did this for the buffer length.

Refactoring Examples

Before:

```
if (time() - start > 86400) {
    [...]
}
```

After:

```
const int kSecondsPerDay = 86400;
if (time() - start > kSecondsPerDay) {
    [...]
}
```

Or:

```
const int kJobWaitSeconds = 60 * 60 * 24;
if (time() - start > kJobWaitSeconds) {
    [...]
}
```

- Replace magic number with symbolic constant.
- There is nothing worse than random numbers strewn around code that have no clear documentation (i.e., why'd they pick that number instead of another?!).
- Instead, use symbolic constants to make the code self-documenting.
- We did this for the buffer length.

Refactoring Examples

Before:

```
const int base_price = quantity_ * item_price_;
const int discount_level = GetDiscountLevel();
const double final_price =
    DiscountedPrice(base_price, discount_level);
```

- Replace param with method is a complex way of describing an encapsulation fix.
- In general, if a value is only used by a function you call, perhaps just have that function compute the value.
- Typically cleans up the caller.
- Doesn't always work; e.g., if the caller is a class member function and the callee isn't.

Refactoring Examples

Before:

```
const int base_price = quantity_ * item_price_;
const int discount_level = GetDiscountLevel();
const double final_price =
    DiscountedPrice(base_price, discount_level);
```

After:

```
const int base_price = quantity_ * item_price_;
const double final_price =
    DiscountedPrice(base_price);

double DiscountedPrice(const int base_price) {
    const int discount_level = GetDiscountLevel());
    [...]
}
```

- Replace param with method is a complex way of describing an encapsulation fix.
- In general, if a value is only used by a function you call, perhaps just have that function compute the value.
- Typically cleans up the caller.
- Doesn't always work; e.g., if the caller is a class member function and the callee isn't.

Before:

```
HttpServer(int port) {
    HandlerMap* handlers = GetHandlers();
    if (handlers == nullptr) {
        // What do I do now?
    }
    [...]
}
```

- Replace Constructor with Factory Method when you want your constructor to be able to fail without using exceptions.*
- Can also help if you want to hide which derived type is being returned based on the input values.
- Potential downside: forces you to allocate this object on the heap. Not usually a huge issue.

*Exceptions in C++ are very difficult to use correctly and often best avoided.

Before:

```
HttpServer(int port) {
    HandlerMap* handlers = GetHandlers();
    if (handlers == nullptr) {
        // What do I do now?
    }
    [...]
}
```

After:

```
HttpServer {
 public:
    [...]
 private:
    HttpServer(int port);  // Hide the c-tor.
}

HttpServer* MakeHttpServer(int port) {
    HandlerMap* handlers = GetHandlers();
    if (handlers == nullptr) {
        return nullptr;
    }
    [...]
}

HttpServer* server = MakeHttpServer(port);
```

- Replace Constructor with Factory Method when you want your constructor to be able to fail without using exceptions.*
- Can also help if you want to hide which derived type is being returned based on the input values.
- Potential downside: forces you to allocate this object on the heap. Not usually a huge issue.

*Exceptions in C++ are very difficult to use correctly and often best avoided.

Refactoring Examples

Before:

```
HttpServer server;
server.SetPort(8080);
server.SetThreadPool(...);
server.AddHandlers(...);
server.Listen();

server.AddHandlers(...);  // This is invalid.
```

- Introduce expression builder when your object has a required call sequence; ie, first call these setup functions, then you can use the rest of the functions.
- Separate setup / construction functions from the runtime functions.
- More flexible than a factory.

Refactoring Examples

Before:

```
HttpServer server;
server.SetPort(8080);
server.SetThreadPool(...);
server.AddHandlers(...);
server.Listen();

server.AddHandlers(...);  // This is invalid.
```

After:

```
HttpServer* server = HttpServerBuilder()
    .SetPort(8080)
    .SetThreadPool(...)
    .AddHandlers(...)
    .Build();
server->Listen();

server->AddHandlers(...);  // Doesn't compile. Yay!
```

- Introduce expression builder when your object has a required call sequence; ie, first call these setup functions, then you can use the rest of the functions.
- Separate setup / construction functions from the runtime functions.
- More flexible than a factory.

Refactoring Examples In Use

```
void ProcessRequest(tcp::socket* sock) {
    while (true) {
        const int kBufferLength = 1024;
        char data[kBufferLength];

        boost::system::error_code error;
        const size_t length =
            sock->read_some(boost::asio::buffer(data), error);

        HandleRequest(data, length);
    }
}

void HandleRequest(const char* buf, const size_t length) {
    [...]
}
```

Replace magic number with symbolic constant

Extract Method

Introduce param object (possibly)

Refactoring & Testing Recap

- You can be more confident that you didn't break anything if you have tests.
- That is, if the tests still pass, your code is *probably* still working.
- As a result, people will often write tests before starting a refactor of poorly tested code.

Dependency Injection - Basics

But first…

A couple more refactorings / code patterns

Favor composition over polymorphism

```
class HasUniqueId {
  int id;
}

HasUniqueId::HasUniqueId() {
  id = nextId++;
}

class Connection :
    public HasUniqueId {}
```

- Polymorphic solution looks ok, not obviously bad code
- Hard to test alone

Favor composition over polymorphism

```
class Connection {
  IdGenerator idGenerator;
}

Connection::Connection(
    IdGenerator idGenerator) {

  this.idGenerator = idGenerator;
}
```

- Now we depend on IdGenerator interface
- At test time, we can provide any implementation we want for `IdGenerator`

new considered harmful

```
Connection::Connection() {
  this.idGenerator =
      new SlowSecureIdGenerator();
  this.id = idGenerator.nextId();
}
```

- If we create an `IdGenerator` in this constructor, we break encapsulation
- Our `Connection` depends on the specific implementation of `IdGenerator` we're `new`-ing
- Be careful to note that every time you instantiate an object, you introduce a dependency on that implementation

new considered harmful

```
class Connection {
  IdGenerator idGenerator;
}

Connection::Connection(
    IdGenerator idGenerator) {

  this.idGenerator = idGenerator;
}
```

- Hmm, this refactor looks exactly the same as the last one…
- This pattern is called “Dependency Injection”

Dependency Injection

- Design pattern in which your objects **ask** for what they need instead of **retrieving** what they need
- In this way, your objects will depend solely on interfaces, and will be **implementation agnostic**


Dependency injection
Passing parameters to functions

Dependency Injection - From first principles

```
public interface Clock {
  Long getCurrentTimestamp();
}
```

Dependency Injection - From first principles

```
public interface Clock {
  Long getCurrentTimestamp();
}
```

```
public class SystemClock
    implements Clock {
  Long getCurrentTimestamp() {
    // [pseudocode] calls into OS
    return System.getTime();
  }
}
```

Dependency Injection - From first principles

```
public interface Clock {
  Long getCurrentTimestamp();
}
```

```
public class SystemClock
    implements Clock {
  Long getCurrentTimestamp() {
    // [pseudocode] calls into OS
    return System.getTime();
  }
}
```

```
class RandomNumberGenerator {
  RandomNumberGenerator() {
    this.clock = new SystemClock();
  }

  Long getSeed() {
    return clock.getCurrentTimestamp();
  }
}
```

Dependency Injection - From first principles

```
public interface Clock {
  Long getCurrentTimestamp();
}
```

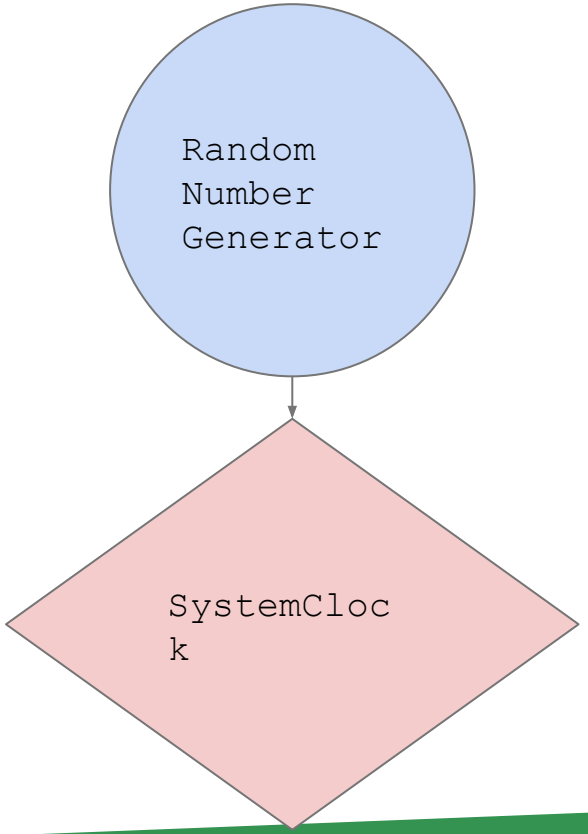
```
public class SystemClock
    implements Clock {
  Long getCurrentTimestamp() {
    // [pseudocode] calls into OS
    return System.getTime();
  }
}
```

```
class RandomNumberGenerator {
  RandomNumberGenerator() {
    this.clock = new SystemClock();
  }

  Long getSeed() {
    return clock.getCurrentTimestamp();
  }
}
```

Does the random number generator really care about what kind of clock it's getting? Or does it just need the `getCurrentTimestamp` method?

Dependency Injection - From principles



Dependency Injection - From first principles

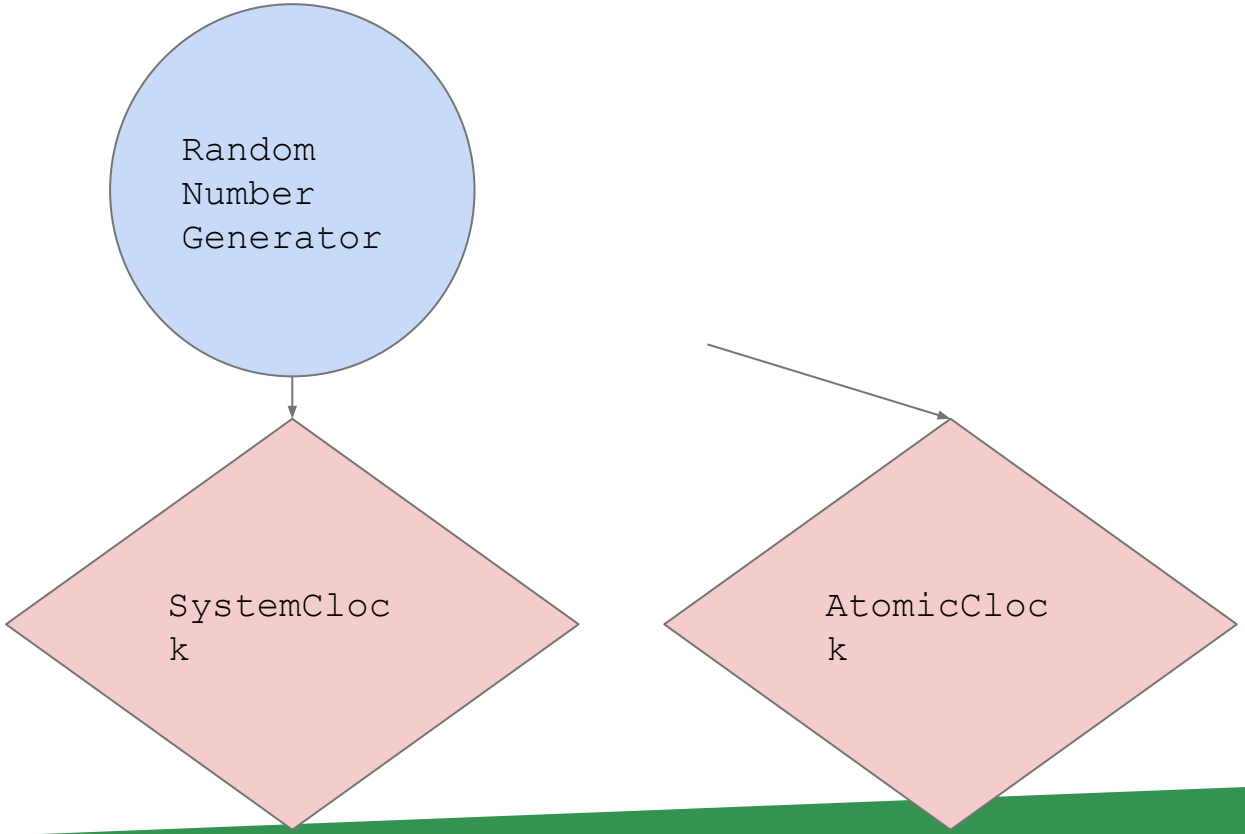
```
public interface Clock {
  Long getCurrentTimestamp();
}
```

```
public class SystemClock
    implements Clock {
  Long getCurrentTimestamp() {
    // [pseudocode] calls into OS
    return System.getTime();
  }
}
```

```
public class AtomicClock
    implements Clock {
  Long getCurrentTimestamp() {
    // [pseudocode] calls internet API
    return InternetClockAPI.getTime();
  }
}
```

Uh oh. Should we use the fancy atomic clock instead?
Let's refactor

Dependency Injection - From principles



Dependency Injection - The pattern

```
public interface Clock {
  Long getCurrentTimestamp();
}
```

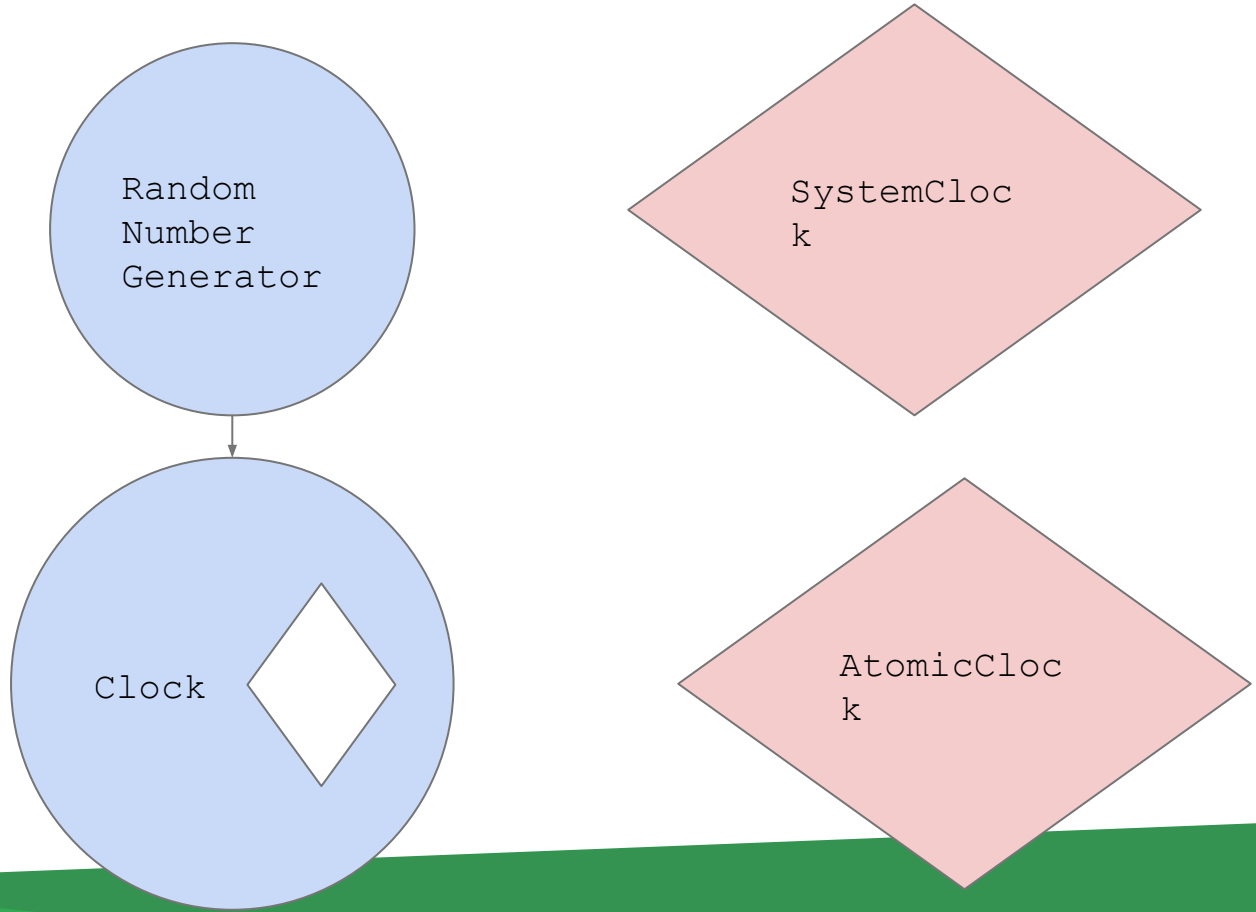
```
public class SystemClock
    implements Clock {
  Long getCurrentTimestamp() {
    // [pseudocode] calls into OS
    return System.getTime();
  }
}
```

```
class RandomNumberGenerator {
  RandomNumberGenerator(Clock clock) {
    this.clock = clock;
  }

  Long getSeed() {
    return clock.getCurrentTimestamp();
  }
}
```

Now `RandomNumberGenerator` does not depend on a specific implementation of `Clock`! Could pass in any `Clock` we want!

Dependency Injection - From principles



Dependency Injection - The pattern

```
public static void main() {
  RandomNumberGenerator rng =
    new RandomNumberGenerator(new SystemClock());
}
```

```
public class SystemClock
    implements Clock {
  Long getCurrentTimestamp() {
    // [pseudocode] calls into OS
    return System.getTime();
  }
}
```

```
class RandomNumberGenerator {
  RandomNumberGenerator(Clock clock) {
    this.clock = clock;
  }

  Long getSeed() {
    return clock.getCurrentTimestamp();
  }
}
```

Dependency Injection - The pattern

```
public static void main() {
  RandomNumberGenerator rng =
    new RandomNumberGenerator(new SystemClock());
}
```

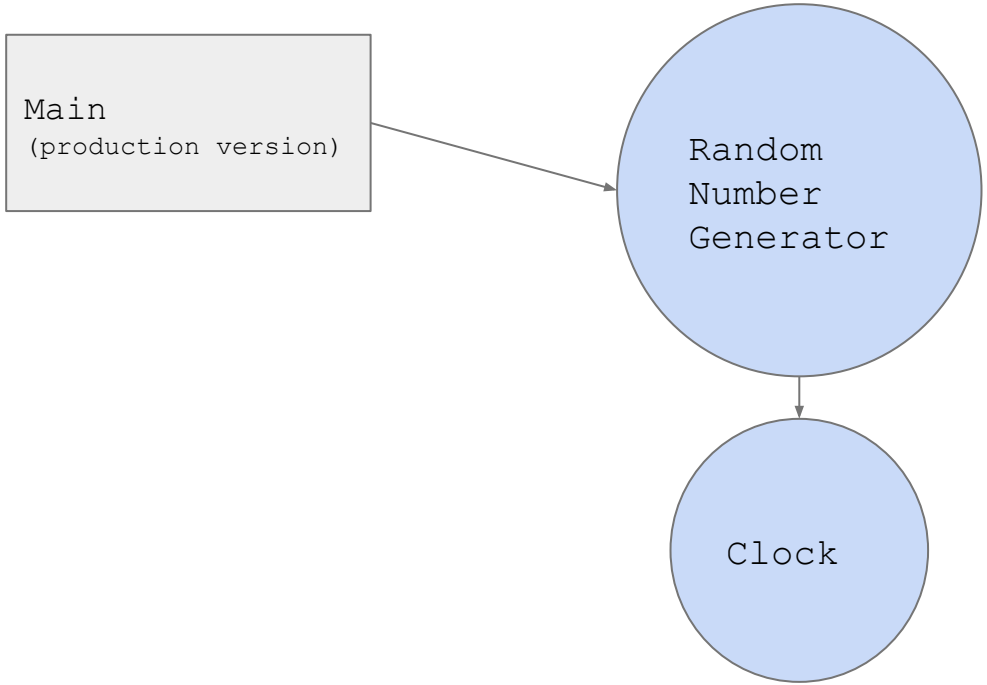
Now, all of our object construction happens in main? Why is that better?

```
public class SystemClock
    implements Clock {
  Long getCurrentTimestamp() {
    // [pseudocode] calls into OS
    return System.getTime();
  }
}
```

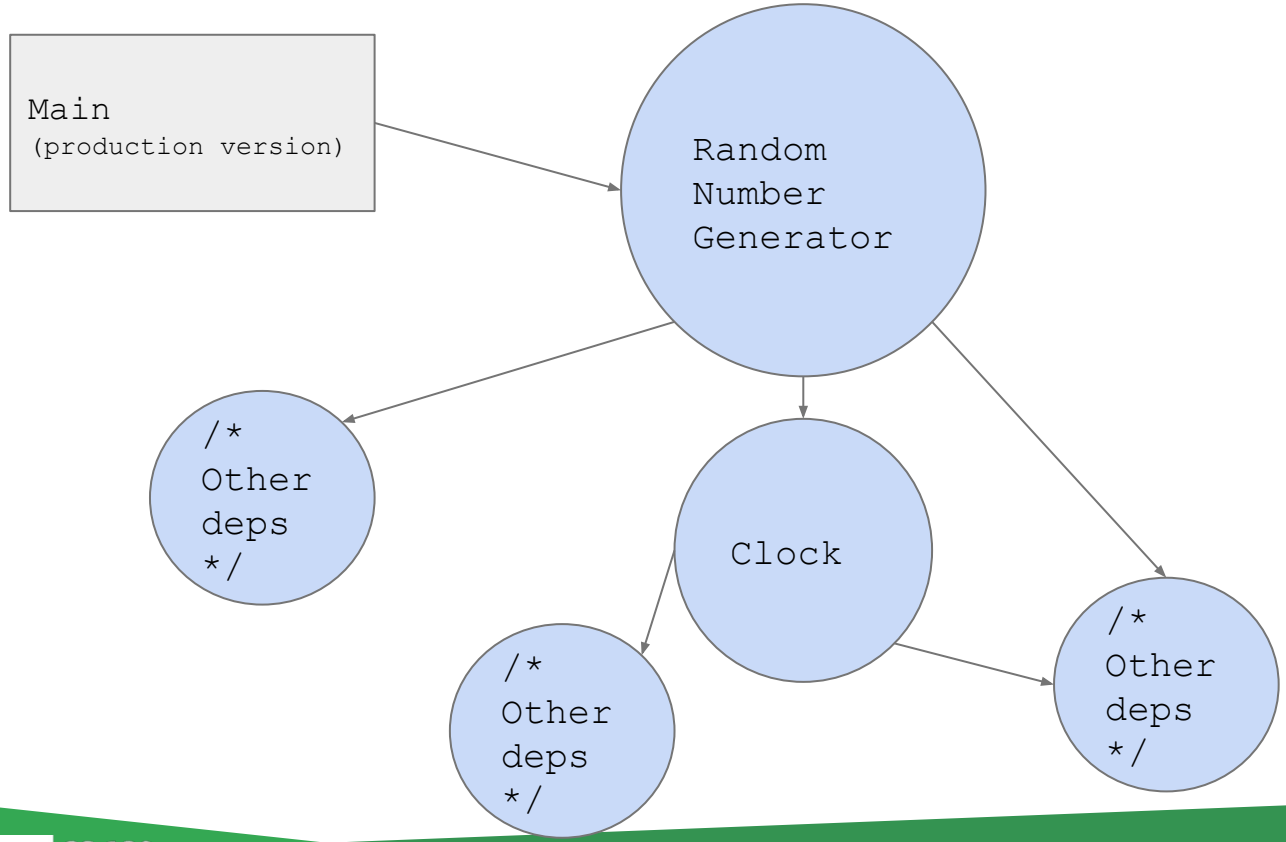
```
class RandomNumberGenerator {
  RandomNumberGenerator(Clock clock) {
    this.clock = clock;
  }

  Long getSeed() {
    return clock.getCurrentTimestamp();
  }
}
```

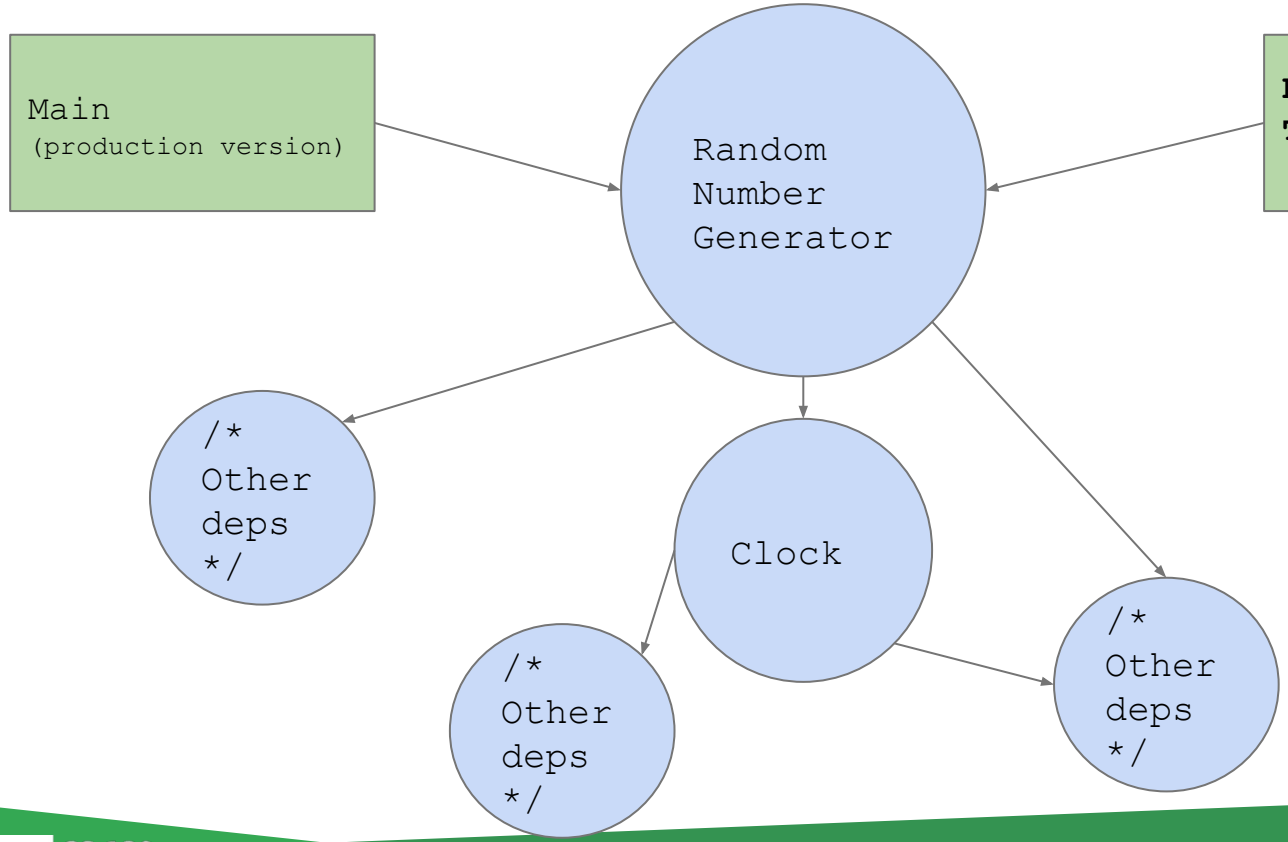
Dependency Injection - Your OO program as a tree



Dependency Injection - Your OO program as a ~~tree~~ graph



Dependency Injection - Your OO program as a ~~tree~~ graph

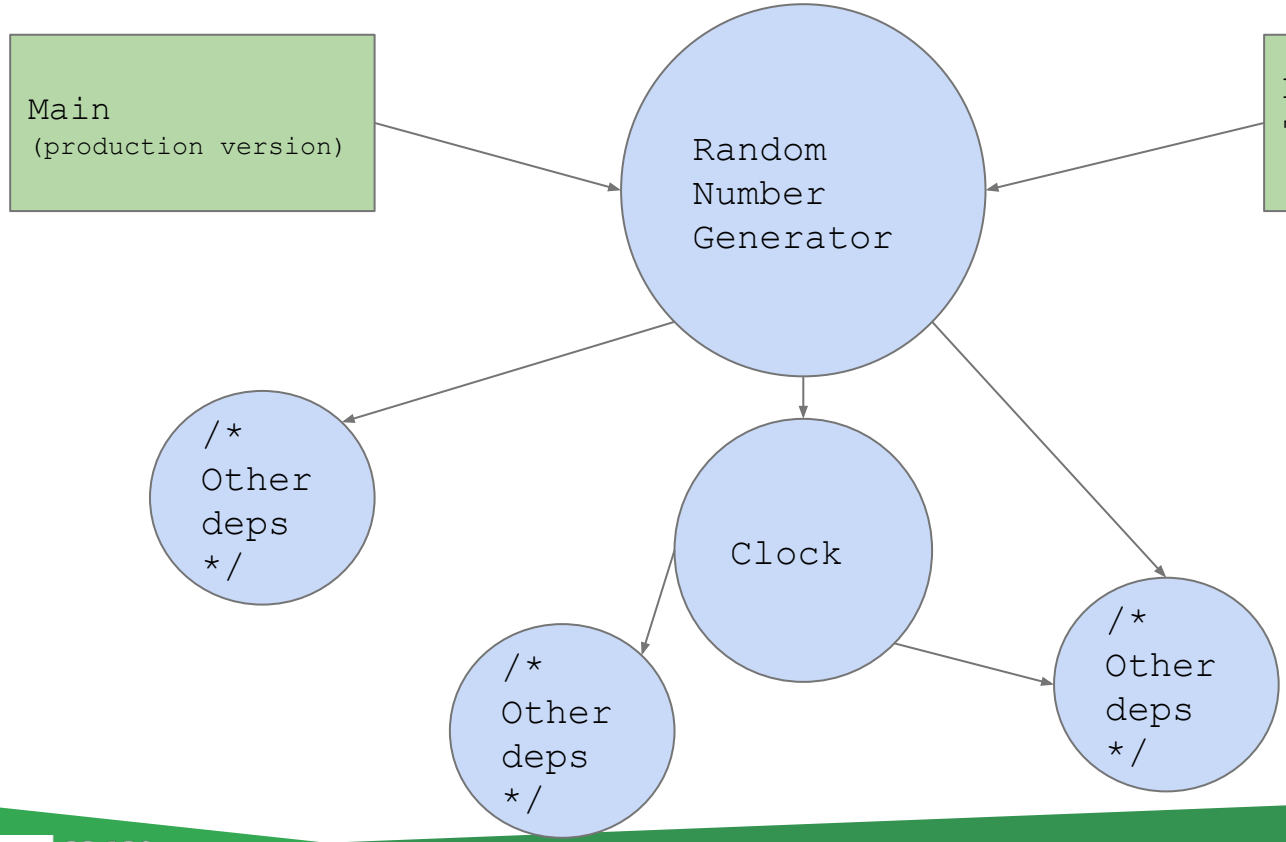


Now we have a reason to move all concrete object construction up to main!

We want to

1. Use different dependency implementations in production vs in unit test
2. Use the same exact class source code in production vs unit test

Dependency Injection - Your OO program as a ~~tree~~ graph

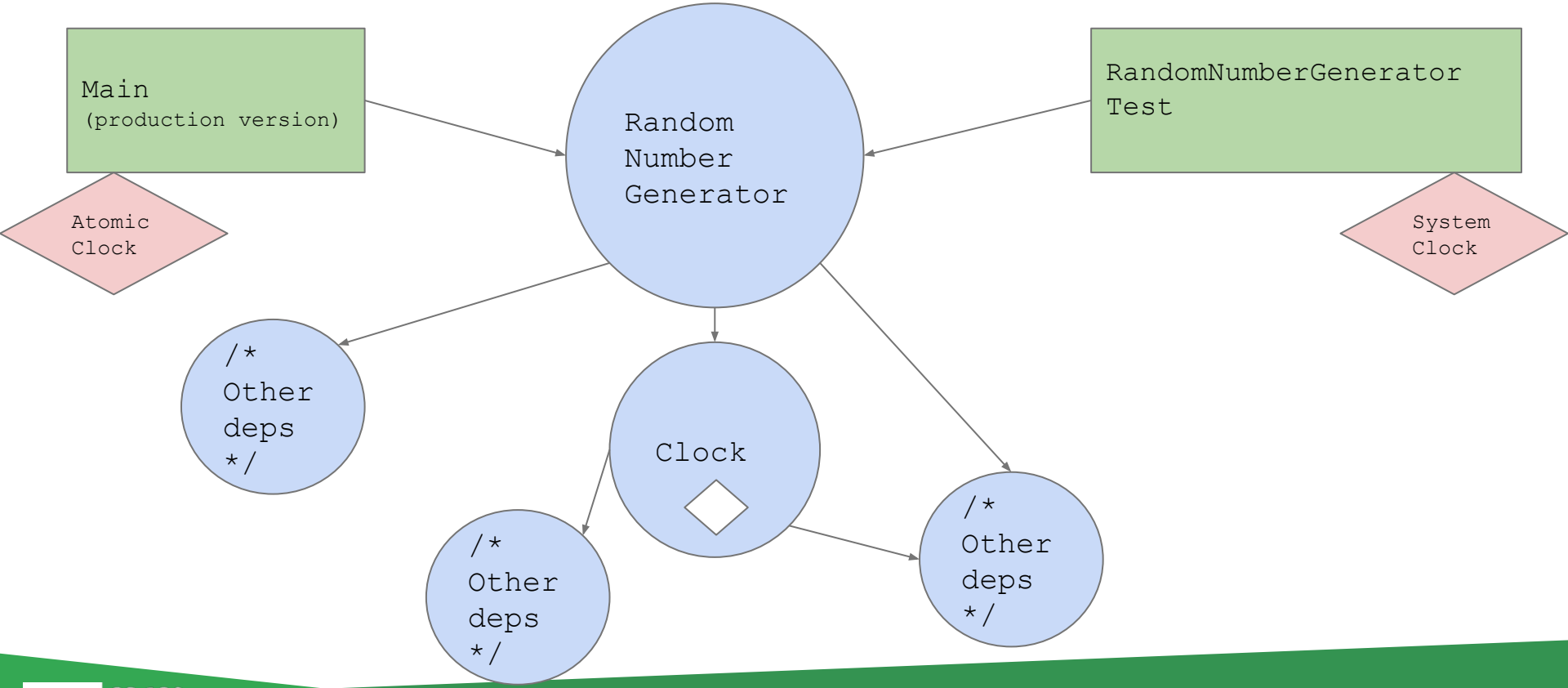


All of the blue nodes are written purely in terms of interfaces

Green nodes determine which concrete implementations to use

Blue nodes can have exactly the same source regardless of which implementations they get at runtime!

Dependency Injection - Your OO program as a ~~tree~~ graph



Dependency Injection - Consider a framework

```
public static void main() {
  RandomNumberGenerator rng =
    new RandomNumberGenerator(new SystemClock());

 /*
 * Instantiate 100s of concrete implementations???
 */
}
```

Dependency Injection - Consider a framework

```
public static void main() {
  RandomNumberGenerator rng =
    new RandomNumberGenerator(new SystemClock());

 /*
 * Instantiate 100s of concrete implementations???
 */
}
```

In reality, your program will likely contain a ton of different types of objects. I have to instantiate them all in the constructor??

This sounds painful. There's got to be a better way!

Next time…

DI Frameworks

https://bit.ly/3xnkD0r

A word: How much do you control your program?

A tweet: Forget the server, how would you refactor your life?