Principles of Software Construction: Objects, Design, and Concurrency

The Last One: Locking Back & Looking Forward

Christian Kästner  Vincent Hellendoorn
Looking Back at the Semester:
194 slides from 23 lectures in 40 min
Principles of Software Construction: Objects, Design, and Concurrency

Introduction, Overview, and Syllabus

Christian Kästner    Vincent Hellendoorn
Welcome to the era of “big code”

Software Size (million Lines of Code)

- Modern High-end Car
- Facebook
- Windows Vista
- Large Hadron Collider
- Boeing 787
- Android
- Google Chrome
- Linux Kernel 2.6.0
- Mars Curiosity Rover
- Hubble Space Telescope
- F-22 Raptor
- Space Shuttle

(informal reports)
Modern Software Engineering

Nobody wants to write a million lines of code.

- Instead, you use libraries
  - E.g., import Android => +12M LOC
  - You don’t write most of the code you use
    - And why would you want to?

- And your libraries use libraries
  - Et cetera
  - https://npm.anvaka.com/#/view/2d/gatsby
From Programs to Applications and Systems

Writing algorithms, data structures from scratch → Reuse of libraries, frameworks

Functions with inputs and outputs → Asynchronous and reactive designs

Sequential and local computation → Parallel and distributed computation

Full functional specifications → Partial, composable, targeted models

Our goal: understanding both the building blocks and also the design principles for construction of software systems at scale
User needs (Requirements)

Miracle?

Code

Maintainable?
Testable?
Extensible?
Scalable?
Robust? ...
Semester overview

- Introduction to Object-Oriented Programming
- Introduction to design
  - Design goals, principles, patterns
- Designing objects/classes
  - Design for change
  - Design for reuse
- Designing (sub)systems
  - Design for robustness
  - Design for change (cont.)
- Design for large-scale reuse

Crosscutting topics:
- Building on libraries and frameworks
- Building libraries and frameworks
- Modern development tools: IDEs, version control, refactoring, build and test automation, static analysis
- Testing, testing, testing
- Concurrency basics
Which version is better?

Version A:

```java
static void sort(int[] list, boolean ascending) {
    ...
    boolean mustSwap;
    if (ascending) {
        mustSwap = list[i] > list[j];
    } else {
        mustSwap = list[i] < list[j];
    }
    ...
}
```

Version B':

```java
interface Order {
    boolean lessThan(int i, int j);
}
class AscendingOrder implements Order {
    public boolean lessThan(int i, int j) { return i < j; }
}
class DescendingOrder implements Order {
    public boolean lessThan(int i, int j) { return i > j; }
}
static void sort(int[] list, Order order) {
    ...
    boolean mustSwap =
        order.lessThan(list[j], list[i]);
    ...
```
it depends

Depends on what?  
What are scenarios?  
What are tradeoffs?

In this specific case, what would you recommend?  
(Engineering judgement)
Some qualities of interest, i.e., design goals

<table>
<thead>
<tr>
<th>Quality</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional</td>
<td>Correctness</td>
</tr>
<tr>
<td>Robustness</td>
<td>Ability to handle anomalous events</td>
</tr>
<tr>
<td>Flexibility</td>
<td>Ability to accommodate changes in specifications</td>
</tr>
<tr>
<td>Reusability</td>
<td>Ability to be reused in another application</td>
</tr>
<tr>
<td>Efficiency</td>
<td>Satisfaction of speed and storage requirements</td>
</tr>
<tr>
<td>Scalability</td>
<td>Ability to serve as the basis of a larger version of the application</td>
</tr>
<tr>
<td>Security</td>
<td>Level of consideration of application security</td>
</tr>
</tbody>
</table>

Source: Braude, Bernstein, Software Engineering. Wiley 2011
Trying to get back to normal with … *
*gestures widely* everything

Talk to us about concerns and accommodations
Disclaimer:
This semester, we are changing a lot in this course. Some things will go wrong. Have patience with us. Give us feedback.

CC BY-NC-ND 2.0 Suzanne Hamilton
Principles of Software Construction
(Design for change, class level)

Starting with Objects
(dynamic dispatch, encapsulation)

Christian Kästner     Vincent Hellendoorn
Where we are

<table>
<thead>
<tr>
<th>Small scale: One/few objects</th>
<th>Mid scale: Many objects</th>
<th>Large scale: Subsystems</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Subtype</strong></td>
<td><strong>Domain Analysis ✓</strong></td>
<td><strong>GUI vs Core ✓</strong></td>
</tr>
<tr>
<td>Polymorphism ✓</td>
<td><strong>Inheritance &amp; Del. ✓</strong></td>
<td><strong>Frameworks and</strong></td>
</tr>
<tr>
<td>Information Hiding,</td>
<td><strong>Responsibility</strong></td>
<td><strong>Libraries ✓, APIs ✓</strong></td>
</tr>
<tr>
<td>Contracts ✓</td>
<td><strong>Assignment,</strong></td>
<td><strong>Module systems,</strong></td>
</tr>
<tr>
<td>Immutability ✓</td>
<td><strong>Design Patterns,</strong></td>
<td><strong>microservices ✓</strong></td>
</tr>
<tr>
<td>Types ✓</td>
<td><strong>Antipattern ✓</strong></td>
<td><strong>Testing for</strong></td>
</tr>
<tr>
<td>Static Analysis ✓</td>
<td><strong>Promises/</strong></td>
<td><strong>Robustness ✓</strong></td>
</tr>
<tr>
<td>Unit Testing ✓</td>
<td><strong>Reactive P. ✓</strong></td>
<td><strong>CI ✓, DevOps ✓, Teams</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Integration Testing ✓</strong></td>
<td></td>
</tr>
</tbody>
</table>

Design for understanding change/ext. reuse robustness...

Small scale:
- One/few objects

Mid scale:
- Many objects

Large scale:
- Subsystems

- GUI vs Core ✓
- Frameworks and Libraries ✓, APIs ✓
- Module systems, microservices ✓
- Testing for Robustness ✓
- CI ✓, DevOps ✓, Teams

Small scale:
- Design for understanding, change/ext., reuse, robustness...

Mid scale:
- Small scale +
- Many objects

Large scale:
- Mid scale +
- Subsystems

- Small scale +
- Mid scale +
- Large scale +
Today: How Objects Respond to Messages

**d: Drawing**
- shapes: Shape[]
- draw(Canvas)

**s0: Square**
- x, y, w, h: int
- draw(Canvas)
- move(int, int)
- ...

**s1: Line**
- from, to: Point
- draw(Canvas)
- move(int, int)
- getLength()
- ...

```java
class Square {
    int x, y, w, h;
    public void draw(Canvas canvas) {
        // draw code
    }
    public void move(int dx, int dy) {
        // move code
    }
}
```
Interface declared explicitly with TypeScript

```typescript
interface Counter {
  v: number;
  inc(): void;
  get(): number;
  add(y: number): number;
}
const obj: Counter = {
  v: 1,
  inc: function() { this.v++; },
  get: function() { return this.v; },
  add: function(y) { return this.v + y; }
}
obj.foo();
// Compile-time error: Property 'foo' does not exist
```

`v` must be part of the interface in TypeScript. Ways to avoid this later.

The object assigned to `obj` must have all the same methods as the interface.
Multiple Implementations of Interface

```java
interface Point {
    int getX();
    int getY();
}

class PolarPoint implements Point {
    double len, angle;
    PolarPoint(double len, double angle) {
        this.len=len; this.angle=angle;
    }
    int getX() { return this.len * cos(this.angle); }  
    int getY() { return this.len * sin(this.angle); }  
    double getAngle() {...}
}

Point p = new PolarPoint(5, .245);
```
How to hide information?

class CartesianPoint {
    int x,y;
    Point(int x, int y) {
        this.x=x;
        this.y=y;
    }
    int getX() { return this.x; }
    int getY() { return this.y; }
    int helper_getAngle();
}

const point = {
    x: 1, y: 0,
    getX: function() {...}
    helper_getAngle: 
        function() {...}
}
Principles of Software Construction: Objects, Design, and Concurrency

IDEs, Build system, Continuous Integration, Libraries

Christian Kästner      Vincent Hellendoorn
## Where we are

<table>
<thead>
<tr>
<th>Small scale: One/few objects</th>
<th>Mid scale: Many objects</th>
<th>Large scale: Subsystems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtype ✓</td>
<td>Domain Analysis ✓</td>
<td>GUI vs Core ✓</td>
</tr>
<tr>
<td>Polymorphism ✓</td>
<td>Inheritance &amp; Del. ✓</td>
<td>Frameworks and Libraries ✓, APIs ✓</td>
</tr>
<tr>
<td>Information Hiding, Contracts ✓</td>
<td>Responsibility Assignment, Design Patterns, Antipattern ✓</td>
<td>Module systems, microservices ✓</td>
</tr>
<tr>
<td>Immutability ✓</td>
<td>Promises/Reactive P. ✓</td>
<td>Testing for Robustness ✓</td>
</tr>
<tr>
<td>Types ✓</td>
<td>Integration Testing ✓</td>
<td>CI ✓, DevOps ✓, Teams</td>
</tr>
<tr>
<td>Static Analysis ✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unit Testing ✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Design for
- understanding
- change/ext.
- reuse
- robustness

...
IDEs

Automate common programming actions:

- Handy refactorings, suggestions
  - E.g., just press `alt+enter` in IntelliJ while highlighting nearly any code
    - Keyboard shortcuts are super useful: explore your IDE!
  - The IDE will refactor the code, and you end up with a lot of
Build Systems

● These days: intricately tied with IDEs, package managers
● Projects often come with a build config file or two
  ○ ‘pom.xml’ for Maven
  ○ ‘tsconfig.json’ + ‘package.json’ for TypeScript+NPM -- the second deals with packages
  ○ These can be nested, one per (sub-)directory, to compose larger systems
    ■ On GitHub, you can create links across repositories
  ○ Specifies:
    ■ Compilation source and target version
    ■ High-level configuration options
    ■ Targets for various phases in development
      ● “lifecycle” in Maven; e.g. ‘compile’, ‘test’, ‘deploy’
    ■ Often involving plugins
    ■ Dependencies with versions
      ● Not shown: in package.json

```json
1  {  
2    "compilerOptions": {  
3      "target": "es2016",  
4      "module": "commonjs",  
5      "sourceMap": true,  
6      "strict": true,  
7      "esModuleInterop": true,  
8      "moduleResolution": "node",  
9      "outDir": "dist"  
10    }  
11  }
```
Continuous integration – Travis CI

Automatically builds, tests, and displays the result
HW1: Extending the Flash Card System
Principles of Software Construction: Objects, Design, and Concurrency

Specifications and unit testing, exceptions

Christian Kästner    Vincent Hellendoorn
Where we are

<table>
<thead>
<tr>
<th>Small scale: One/few objects</th>
<th>Mid scale: Many objects</th>
<th>Large scale: Subsystems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtype Polymorphism ✓</td>
<td>Domain Analysis ✓</td>
<td>GUI vs Core ✓</td>
</tr>
<tr>
<td>Information Hiding, <strong>Contracts</strong> ✓</td>
<td>Inheritance &amp; Del. ✓</td>
<td>Frameworks and Libraries ✓, APIs ✓</td>
</tr>
<tr>
<td>Immutability ✓</td>
<td>Responsibility Assignment, Design Patterns, Antipattern ✓</td>
<td>Module systems, microservices ✓</td>
</tr>
<tr>
<td>Types ✓</td>
<td>Promises/ Reactive P. ✓</td>
<td>Testing for Robustness ✓</td>
</tr>
<tr>
<td>Static Analysis ✓</td>
<td>Integration Testing ✓</td>
<td>CI ✓, DevOps ✓, Teams</td>
</tr>
<tr>
<td><strong>Unit Testing ✓</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Design for understanding change/ext. reuse robustness ...

Small scale:
- One/few objects

Mid scale:
- Many objects
  - Domain Analysis ✓
  - Inheritance & Del. ✓
  - Responsibility Assignment, Design Patterns, Antipattern ✓
  - Promises/ Reactive P. ✓
  - Integration Testing ✓

Large scale:
- Subsystems
  - GUI vs Core ✓
  - Frameworks and Libraries ✓, APIs ✓
  - Module systems, microservices ✓
  - Testing for Robustness ✓
  - CI ✓, DevOps ✓, Teams
Handling Exceptions

```java
String read(String path) {
    try {
        return Files.lines(Path.of(path))
            .collect(Collectors.joining("\n"));
    } catch (IOException e) {
        // implement fall-back behavior.
    }
}
```
Testing

How do we know this works?

```java
int isPos(int x) {
    return x >= 0;  // What if?
}

@Test
void test1IsPos() {
    assertTrue(isPos(1));
}

@Test
void test0IsNotPos() {
    assertFalse(isPos(0));  // Fails
}
```

Are we done?
class RepeatingCardOrganizer {
...
/**
 * Checks if the provided card has been answered correctly the required number of times.
 * @param card The {@link CardStatus} object to check.
 * @return {@code true} if this card has been answered correctly at least
 * {@code this.repetitions} times.
 */
  public boolean isComplete(CardStatus card) {
    // IGNORE THIS WHEN SPECIFICATION TESTING!
  }
}
Test case design
Where we are

<table>
<thead>
<tr>
<th>Design for understanding change/ext.</th>
<th>Small scale: One/few objects</th>
<th>Mid scale: Many objects</th>
<th>Large scale: Subsystems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design for understanding change/ext.</td>
<td>Subtype Polymorphism ✓</td>
<td>Domain Analysis ✓</td>
<td>GUI vs Core ✓</td>
</tr>
<tr>
<td>change/ext.</td>
<td>Information Hiding, Contracts ✓</td>
<td>Inheritance &amp; Del. ✓</td>
<td>Frameworks and Libraries ✓, APIs ✓</td>
</tr>
<tr>
<td>reuse</td>
<td>Immutability ✓</td>
<td>Responsibility Assignment, Design Patterns, Antipattern ✓</td>
<td>Module systems, microservices ✓</td>
</tr>
<tr>
<td>reuse</td>
<td>Types ✓</td>
<td>Promises/Reactive P. ✓</td>
<td>Testing for Robustness ✓</td>
</tr>
<tr>
<td>robustness</td>
<td>Static Analysis ✓</td>
<td>Integration Testing ✓</td>
<td>CI ✓, DevOps ✓, Teams</td>
</tr>
<tr>
<td>...</td>
<td>Unit Testing ✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
public boolean pay(int cost, boolean useCredit) {
    if (useCredit) {
        if (enoughCredit) {
            return true;
        }
    }
    if (enoughCash) {
        return true;
    }
    return false;
}

<table>
<thead>
<tr>
<th>Test case</th>
<th>useCredit</th>
<th>Enough Credit</th>
<th>Enough Cash</th>
<th>Result</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>T</td>
<td>-</td>
<td>Pass</td>
<td>--</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>-</td>
<td>T</td>
<td>Pass</td>
<td>--</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>-</td>
<td>F</td>
<td>Fails</td>
<td>Statement</td>
</tr>
</tbody>
</table>
Structures in Code

sequence

if .. then

If .. then .. else

Do .. While

While .. Do

Switch
Writing Testable Code

Aim to write easily testable code

● Which is almost by definition more modular

```java
public List<String> getLines(String path) throws IOException {
    return Files.readAllLines(Path.of(path));
}

public boolean hasHeader(List<String> lines) {
    return !lines.get(0).isEmpty()
}

// Test:
// - hasHeader with empty, non-empty first line
// - getLines (if you must) with null, real path
```
Boundary Value Testing

We need a *strategy* to identify plausible mistakes

- **Boundary Value Testing**: errors often occur at boundary conditions
  - Select: a nominal/normal case, a boundary value, and an abnormal case
  - Useful for few *categories* of behavior (e.g., null/not-null) per value
- **Test**: cost < credit, cost == credit, cost > credit, cost < cash, cost == cash, cost > cash

```java
/** Pays with credit if useCredit is set and enough
 * credit is available; otherwise, pays with cash if
 * enough cash is available; otherwise, returns false.
 */
public boolean pay(int cost, boolean useCredit);
```
HW 2: Testing the Flash Card System
Principles of Software Construction: Objects, Design, and Concurrency

Object-oriented analysis

Christian Kästner  Vincent Hellendoorn
Where we are

<table>
<thead>
<tr>
<th>Small scale: One/few objects</th>
<th>Mid scale: Many objects</th>
<th>Large scale: Subsystems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtype</td>
<td>Domain Analysis ✓</td>
<td>GUI vs Core ✓</td>
</tr>
<tr>
<td>Polymorphism ✓</td>
<td>Inheritance &amp; Del. ✓</td>
<td>Frameworks and Libraries ✓, APIs ✓</td>
</tr>
<tr>
<td>Information Hiding, Contracts ✓</td>
<td>Responsibility Assignment, Design Patterns, Antipattern ✓</td>
<td>Module systems, microservices ✓</td>
</tr>
<tr>
<td>Immutability ✓</td>
<td>Promises/ Reactive P. ✓</td>
<td>Testing for Robustness ✓</td>
</tr>
<tr>
<td>Types ✓</td>
<td>Integration Testing ✓</td>
<td>CI ✓, DevOps ✓, Teams</td>
</tr>
<tr>
<td>Static Analysis ✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unit Testing ✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Design for**
understanding
change/ext.
reuse
robustness
...

Small scale:
- One/few objects

Mid scale:
- Many objects

Large scale:
- Subsystems
Lufthansa Flight 2904

- The Airbus A320-200 airplane has a software-based braking system
- Engaging reverse thrusters while in the air is very dangerous: Only allow breaking when on the ground
**Problem Space**
(Domain Model)
- Real-world concepts
- Requirements, Concepts
- Relationships among concepts
- Solving a problem
- Building a vocabulary

**Solution Space**
(Object Model)
- System implementation
- Classes, objects
- References among objects and inheritance hierarchies
- Computing a result
- Finding a solution
An object-oriented design process

Model / diagram the problem, define concepts
- **Domain model** (a.k.a. conceptual model), **glossary**

Define system behaviors
- **System sequence diagram**
- **System behavioral contracts**

Assign object responsibilities, define interactions
- **Object interaction diagrams**

Model / diagram a potential solution
- **Object model**

**OO Analysis:**
Understanding the problem

**OO Design:**
Defining a solution
Visual notation: UML

Name of real-world concept (not software class)

Library Account
- accountID
- lateFees

Book
- title
- author

Associations between concepts
Multiplicities/cardinalities indicate “how many”
One domain model for the library system
UML Sequence Diagram Notation

Actors in this use case (systems and real-world objects/people)

Time proceeds from top to bottom

Messages and responses for interactions, text describes what happens conceptually
UML Sequence Diagram Notation

Actors in this use case (systems and real-world objects/people)

Time proceeds from top to bottom

Messages and responses for interactions, text describes what happens conceptually
Formalize system at boundary

A system behavioral contract describes the pre-conditions and post-conditions for some operation identified in the system sequence diagrams

- System-level textual specifications, like software specifications
Assigning Responsibilities

Christian Kästner  Vincent Hellendoorn
Where we are

<table>
<thead>
<tr>
<th>Design for understanding</th>
<th>Small scale: One/few objects</th>
<th>Mid scale: Many objects</th>
<th>Large scale: Subsystems</th>
</tr>
</thead>
<tbody>
<tr>
<td>change/ext.</td>
<td>Subtype Polymorphism ✓</td>
<td>Domain Analysis ✓</td>
<td>GUI vs Core ✓</td>
</tr>
<tr>
<td>reuse</td>
<td>Information Hiding, Contracts ✓</td>
<td>Inheritance &amp; Del. ✓</td>
<td>Frameworks and Libraries ✓, APIs ✓</td>
</tr>
<tr>
<td>robustness</td>
<td>Immutability ✓</td>
<td>Responsibility Assignment, Design Patterns, Antipattern ✓</td>
<td>Module systems, microservices ✓</td>
</tr>
<tr>
<td></td>
<td>Types ✓</td>
<td>Promises/Reactive P. ✓</td>
<td>Testing for Robustness ✓</td>
</tr>
<tr>
<td></td>
<td>Static Analysis ✓</td>
<td>Integration Testing ✓</td>
<td>CI ✓, DevOps ✓, Teams</td>
</tr>
<tr>
<td></td>
<td>Unit Testing ✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Object Diagrams

Objects/classes with fields and methods

Interfaces with methods

Associations, visibility, types
always start with an initial method

:Library System

loginMember(libraryCard)

retrieveAccount(libraryCard, idNumber)

account

setCurrentSession(account)

:SessionManager
Doing and Knowing *Responsibilities*

Responsibilities are related to the obligations of an object in terms of its behavior.

Doing responsibilities of an object include:
- doing something itself, such as creating an object or doing a calculation
- initiating action in other objects
- controlling and coordinating activities in other objects

Knowing responsibilities of an object include:
- knowing about private encapsulated data
- knowing about related objects
- knowing about things it can derive or calculate
Low Representational Gap

Identified concepts provide inspiration for classes in the implementation

Classes mirroring domain concepts often intuitive to understand, rarely change (low representational gap)

class LibraryDatabase {
    Map<Int, List<Int>> borrowedBookIds;
    Map<Int, Int> late Fees;
    Map<Int, String> bookTitles;
}

class DatabaseRow { ... }
Which classes are coupled?
How can coupling be improved?

class Shipment {
    private List<Box> boxes;
    int getWeight() {
        int w = 0;
        for (Box box : boxes) {
            for (Item item : box.getItems())
                w += item.weight;
        }
        return w;
    }
}
class Box {
    private List<Item> items;
    Iterable<Item> getItems() {
        return items;
    }
}
class Item {
    Box containedIn;
    int weight;
}
Anti-Pattern: God Object

class Chat {
    List<String> channels;
    Map<String, List<Msg>> messages;
    Map<String, String> accounts;
    Set<String> bannedUsers;
    File logFile;
    File bannedWords;
    URL serverAddress;
    Map<String, Int> globalSettings;
    Map<String, Int> userSettings;
    Map<String, Graphic> smileys;
    CryptStrategy encryption;
    Widget sendButton, messageList;
}

class ChatUI {
    Chat chat;
    Widget sendButton, ...
}

class AccountMgr {
    ... accounts, bannedUsr...
}
Information Expert ->  "Do It Myself Strategy"

Expert usually leads to designs where a software object does those operations that are normally done to the inanimate real-world thing it represents

- a sale does not tell you its total; it is an inanimate thing

In OO design, all software objects are "alive" or "animated," and they can take on responsibilities and do things.

They do things related to the information they know.
Creator: Discussion of Design Goals/Principles

Promotes **low coupling, high cohesion**
- class responsible for creating objects it needs to reference
- creating the objects themselves avoids depending on another class to create the object

Promotes **evolvability** (design for change)
- Object creation is hidden, can be replaced locally

Contra: sometimes objects must be created in special ways
- complex initialization
- instantiate different classes in different circumstances
- *then cohesion suggests putting creation in a different object*: see design patterns such as builder, factory method
HW3: Santorini (Base game)
Principles of Software Construction: Objects, Design, and Concurrency

Design Patterns

Christian Kästner    Vincent Hellendoorn
## Where we are

<table>
<thead>
<tr>
<th>Subtype Polymorphism ✓</th>
<th>Domain Analysis ✓</th>
<th>GUI vs Core ✓</th>
</tr>
</thead>
<tbody>
<tr>
<td>Information Hiding, Contracts ✓</td>
<td>Inheritance &amp; Del. ✓</td>
<td>Frameworks and Libraries ✓, APIs ✓</td>
</tr>
<tr>
<td>Immutability ✓</td>
<td>Responsibility Assignment, Design Patterns, Antipattern ✓</td>
<td>Module systems, microservices ✓</td>
</tr>
<tr>
<td>Types ✓</td>
<td>Promises/Reactive P. ✓</td>
<td>Testing for Robustness ✓</td>
</tr>
<tr>
<td>Static Analysis ✓</td>
<td>Integration Testing ✓</td>
<td>CI ✓, DevOps ✓, Teams</td>
</tr>
<tr>
<td>Unit Testing ✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Design for understanding
- change/ext.
- reuse
- robustness

...
Discussion with design patterns

● **Carpentry:**
  ○ "Is a dovetail joint or a miter joint better here?"

● **Software Engineering:**
  ○ "Is a strategy pattern or a template method better here?"
History: *Design Patterns* (1994)
Module pattern: Decide what to export

```javascript
var MODULE = (function () {
    var my = {},
        privateVariable = 1;

    function privateMethod() {
        // ...
    }

    my.moduleProperty = 1;
    my.moduleMethod = function () {
        // ...
    };

    return my;

})();
```
The Composite Design Pattern

```java
operation() {
    for (c in children)
        c.operation();
}
```
Principles of Software Construction: Objects, Design, and Concurrency

Inheritance and delegation

Christian Kästner  Vincent Hellendoorn
Where we are

<table>
<thead>
<tr>
<th></th>
<th>Small scale: One/few objects</th>
<th>Mid scale: Many objects</th>
<th>Large scale: Subsystems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design for</td>
<td></td>
<td></td>
<td>GUI vs Core ✓</td>
</tr>
<tr>
<td>understanding</td>
<td></td>
<td></td>
<td>Frameworks and Libraries ✓, APIs ✓</td>
</tr>
<tr>
<td>change/ext.</td>
<td></td>
<td></td>
<td>Module systems, microservices ✓</td>
</tr>
<tr>
<td>reuse</td>
<td></td>
<td></td>
<td>Testing for Robustness ✓</td>
</tr>
<tr>
<td>robustness</td>
<td></td>
<td></td>
<td>CI ✓, DevOps ✓, Teams</td>
</tr>
<tr>
<td></td>
<td>Subtype</td>
<td>Domain Analysis ✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Polymorphism ✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Information Hiding,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Contracts ✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Immutability ✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Types ✓</td>
<td>Responsibility Assignment,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Static Analysis ✓</td>
<td>Design Patterns,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unit Testing ✓</td>
<td>Antipattern ✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Promises/</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reactive P. ✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Integration Testing ✓</td>
<td></td>
</tr>
</tbody>
</table>
Class Hierarchy

In Java:

- `Object`
- `Error`
- `Exception`
- `RuntimeError`
- `Collection`
- `List`
Behavioral Subtyping

- Formalizes notion of extension

```java
Animal dog = new Dog();
```

- Roughly: anything an Animal does, a Dog should do
- You should be able to use a subtype as if it was its parent
- But, dog may be more specific

The Liskov substitution principle:
“Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.”

Barbara Liskov
So why inheritance?

```java
public interface PaymentCard {
    String getCardHolderName();
    BigInteger getDigits();
    Date getExpiration();
    int getValue();
    boolean pay(int amount);
}

class CreditCard implements PaymentCard {
    private final String cardHolderName;
    private final BigInteger digits;
    private final Date expirationDate;
    private final int creditLimit;
    private int currentCredit;

    public CreditCard(String cardHolderName, BigInteger digits, Date expirationDate, int creditLimit, int credit) {
        this.cardHolderName = cardHolderName;
        this.digits = digits;
        this.expirationDate = expirationDate;
        this.creditLimit = creditLimit;
        this.currentCredit = credit;
    }
    
    @Override
    public String getCardHolderName() {
        return this.cardHolderName;
    }
    
    @Override
    public BigInteger getDigits() {
        return this.digits;
    }

    @Override
    public Date getExpiration() {
        return this.expirationDate;
    }
}
```
Template Method Pattern

abstract class AbstractCashCard implements PaymentCard {
    private int balance;
    public AbstractCashCard(int balance) {
        this.balance = balance;
    }
    public boolean pay(int amount) {
        if (amount <= this.balance) {
            this.balance -= amount;
            chargeFee();
            return true;
        }
        return false;
    }
    abstract void chargeFee();
}

class GiftCard extends AbstractCashCard {
    @Override
    void chargeFee() {
        return; // Do nothing.
    }
}

‘Pay’ is already implemented
Template Method vs. Strategy Pattern

- **Template method uses inheritance to vary part of an algorithm**
  - Template method implemented in supertype, primitive operations implemented in subtypes

- **Strategy pattern uses delegation to vary the entire algorithm**
  - Strategy objects are reusable across multiple classes
  - Multiple strategy objects are possible per class
Refactoring

- Rename class, method, variable to something not in-scope
- Extract method/inline method
- Extract interface
- Move method (up, down, laterally)
- Replace duplicates
Principles of Software Construction: Objects, Design, and Concurrency

Refactoring & Anti-patterns

Christian Kästner  Vincent Hellendoorn
## Where we are

<table>
<thead>
<tr>
<th>Design for understanding</th>
<th>Small scale: One/few objects</th>
<th>Mid scale: Many objects</th>
<th>Large scale: Subsystems</th>
</tr>
</thead>
<tbody>
<tr>
<td>change/ext.</td>
<td>Subtype Polymorphism ✓</td>
<td>Domain Analysis ✓</td>
<td>GUI vs Core ✓</td>
</tr>
<tr>
<td>reuse</td>
<td>Information Hiding, Contracts ✓</td>
<td>Inheritance &amp; Del. ✓</td>
<td>Frameworks and Libraries ✓, APIs ✓</td>
</tr>
<tr>
<td>robustness</td>
<td>Immutability ✓</td>
<td>Responsibility Assignment, Design Patterns, <strong>Antipattern ✓</strong></td>
<td>Module systems, microservices ✓</td>
</tr>
<tr>
<td></td>
<td>Types ✓</td>
<td>Promises/Reactive P. ✓</td>
<td>Testing for Robustness ✓</td>
</tr>
<tr>
<td></td>
<td>Static Analysis ✓</td>
<td>Integration Testing ✓</td>
<td>CI ✓, DevOps ✓, Teams</td>
</tr>
<tr>
<td></td>
<td>Unit Testing ✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Decorator Pattern

You have a complex drawing that consists of many shapes and want to save it. Some logic of the saving functionality is always the same (e.g., going through all shapes, reducing them to drawable lines), but others you want to vary to support saving in different file formats (e.g., as png, as svg, as pdf). You want to support different file formats later.

Why is this not:

https://refactoring.guru/design-patterns/decorator
This binding

class Parent {
    private int i;
    public Parent() {
        this.i = 5;
    }
    void print() {
        System.out.println(this.i);
    }
}

class Child extends Parent {
    private int i;
    public Child() {
        this.i = 7;
    }
}

Child m = new Child();
System.out.println(m.i);
m.print();
Details: type-casting

- Sometimes you want a different type than you have
  - e.g., double pi = 3.14;
    int indianaPi = (int) pi;

- Useful if you know you have a more specific subtype:
  Account acct = ...;
  CheckingAccount checkingAcct = (CheckingAccount) acct;
  long fee = checkingAcct.getFee();
  - Will get a ClassCastException if types are incompatible

- Advice: avoid downcasting types
  - Never(?) downcast within superclass to a subclass

In TS:
(dog as Animal).identify()
Anti-patterns

● Zooming in: common code smells
  ○ Not necessarily bad, but worthwhile indicators to check
    ■ When problematic, often point to design problems
  ○ Long methods, large classes, and the likes. Suggests bad abstraction
    ■ Tend to evolve over time; requires restructuring
  ○ Inheritance despite low coupling (“refused bequest”)
    ■ Replace with delegation, or rebalance hierarchy
  ○ ‘instanceof’ (or ‘switch’) instead of polymorphism
  ○ Overly similar classes, hierarchies
  ○ Any change requires lots of edits
    ■ High coupling across classes (“shotgun surgery”), or heavily entangled implementation (intra-class)
HW4: Refactoring of Static Website Generator
Principles of Software Construction: Objects, Design, and Concurrency

Asynchrony and Concurrency

Christian Kästner  Vincent Hellendoorn
Where we are

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

Scanner input = new Scanner(System.in);
while (questions.hasNext()) {
    Question q = question.next();
    System.out.println(q.toString());
    String answer = input.nextLine();
    q.respond(answer);
}
Event-based programming

- Style of programming where control-flow is driven by (usually external) events

```java
public void performAction(ActionEvent e) {
    List<String> lst = Arrays.asList(bar);
    foo.peek(42)
}
```

```java
public void performAction(ActionEvent e) {
    bigBloatedPowerPointFunction(e);
    withANameSoLongIMadeItTwoMethods(e);
    yesIKnowJavaDoesntWorkLikeThat(e);
}
```

```java
public void performAction(ActionEvent e) {
    List<String> lst = Arrays.asList(bar);
    foo.peek(40)
}
```
Concurrency with file I/O

Asynchronous code requires Promises

- Captures an intermediate state
  - Neither fetched, nor failed: we’ll find out eventually

```javascript
let imageToBe: Promise<Image> = fetch('myImage.png');
imageToBe.then((image) => display(image))
  .catch((err) => console.log('aw: ' + err));
```
Concurrency with file I/O

An example from Machine Learning

<table>
<thead>
<tr>
<th>Epoch</th>
<th>Train</th>
<th>Read</th>
<th>Prefetch</th>
<th>Open</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Naive</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Different devices:
A GUI design challenge, extended

- What if we want to show the points won?
Recall the Observer

https://refactoring.guru/design-patterns/observer
An architectural pattern: Model-View-Controller (MVC)

- **Model**: Manage data related to the application domain
- **View**: Manage display of information on the screen
- **Controller**: Manage inputs from user: mouse, keyboard, menu, etc.
Principles of Software Construction: Objects, Design, and Concurrency

Basic GUI concepts, HTML

Christian Kästner  Vincent Hellendoorn
Where we are

<table>
<thead>
<tr>
<th>Design for understanding change/extend reuse robustness</th>
<th>Small scale: One/few objects</th>
<th>Mid scale: Many objects</th>
<th>Large scale: Subsystems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtype Polymorphism ✓</td>
<td>Domain Analysis ✓</td>
<td>GUI vs Core ✓</td>
<td></td>
</tr>
<tr>
<td>Information Hiding, Contracts ✓</td>
<td>Inheritance &amp; Del. ✓</td>
<td>Frameworks and Libraries ✓, APIs ✓</td>
<td></td>
</tr>
<tr>
<td>Immutability ✓</td>
<td>Responsibility Assignment, Design Patterns, Antipattern ✓</td>
<td>Module systems, microservices ✓</td>
<td></td>
</tr>
<tr>
<td>Types ✓</td>
<td>Promises/Reactive P. ✓</td>
<td>Testing for Robustness ✓</td>
<td></td>
</tr>
<tr>
<td>Static Analysis ✓</td>
<td>Integration Testing ✓</td>
<td>CI ✓, DevOps ✓, Teams</td>
<td></td>
</tr>
<tr>
<td>Unit Testing ✓</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...
Anatomy of an HTML Page

Nested elements

- Sizing
- Attributes
- Text
The composite pattern

- **Problem:** Collection of objects has behavior similar to the individual objects
- **Solution:** Have collection of objects and individual objects implement the same interface
- **Consequences:**
  - Client code can treat collection as if it were an individual object
  - Easier to add new object types
  - Design might become too general, interface insufficiently useful
A few Tags

- `<html>`
  - The root of the visible page
- `<head>`
  - Stores metadata, imports
- `<p>`
  - A paragraph
- `<button>`
  - Attributes include `name`, `type`, `value`
- `<div>`
  - Generic section -- very useful
- `<table>`
  - The obvious
- Many more; dig into a real page!
Style: CSS

● Cascading Style Sheets
  ○ Reuse: styling rules for tags, classes, types
  ○ Reuse: not just at the leafs!

```html
<span style="font-weight:bold">Hello again!</span>
```

vs.

```html
<style type="text/css">
  span {
    font-family: arial
  }
</style>
```
Strategy or Observer?

Either could apply

- Both involve callback
- Strategy:
  - Typically single
  - Often involves a return
- Observer:
  - Arbitrarily many
  - Involves external updates
Static Web Pages

- Delivered as-is, final
  - Consistent, often fast
  - Cheap, only storage needed

- “Static” a tad murky with JavaScript
  - We can still have buttons, interaction
  - But it won’t “go” anywhere -- the server is mum
Web Servers

Dynamic sites can do more work

https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Client-Server_overview#anatomy_of_a_dynamic_request
Concurrency: Safety & Immutability

Christian Kästner      Vincent Hellendoorn
Where we are

<table>
<thead>
<tr>
<th>Design for</th>
<th>Small scale: One/few objects</th>
<th>Mid scale: Many objects</th>
<th>Large scale: Subsystems</th>
</tr>
</thead>
<tbody>
<tr>
<td>understanding</td>
<td>Subtype Polymorphism ✓</td>
<td>Domain Analysis ✓</td>
<td>GUI vs Core ✓</td>
</tr>
<tr>
<td>change/ext.</td>
<td>Information Hiding, Contracts ✓</td>
<td>Inheritance &amp; Del. ✓</td>
<td>Frameworks and Libraries ✓, APIs ✓</td>
</tr>
<tr>
<td>reuse</td>
<td>Immutability ✓</td>
<td>Responsibility Assignment, Design Patterns, Antipattern ✓</td>
<td>Module systems, microservices ✓</td>
</tr>
<tr>
<td>robustness</td>
<td>Types ✓</td>
<td>Promises/Reactive P. ✓</td>
<td>Testing for Robustness ✓</td>
</tr>
<tr>
<td>...</td>
<td>Static Analysis ✓</td>
<td>Integration Testing ✓</td>
<td>CI ✓, DevOps ✓, Teams</td>
</tr>
<tr>
<td></td>
<td>Unit Testing ✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Components of a Swing application

- JFrame
- JPanel
- JButton
- JTextField
- ...
Event Loop in JS

Web APIs
- DOM (document)
- AJAX (XMLHttpRequest)
- Timeout (setTimeout)

Memory Heap

Call Stack

Event Loop

Callback Queue
- onClick
- onLoad
- onDone
What will Happen:

Where does this fail?

What if single threaded?

Could we make it work with 2 threads?
Ensuring Immutability

- Don’t provide any mutators
- Ensure that no methods may be overridden
- Make all fields final
- Make all fields private
- Ensure security of any mutable components
Making a Class Immutable

public final class Complex {
    private final double re, im;

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    // Getters without corresponding setters
    public double getRealPart() { return re; }
    public double getImaginaryPart() { return im; }

    // subtract, multiply, divide similar to add
    public Complex add(Complex c) {
        return new Complex(re + c.re, im + c.im);
    }
}
Shared State

- **Volatile** fields always return the most recently written value
  - Does **not** guarantee atomicity
  - Useful if only one thread writes

- Are atomicity + coordinated communication sufficient for thread safety?
Principles of Software Construction: Objects, Design, and Concurrency

Concurrency: Patterns & Promises

Christian Kästner
Vincent Hellendoorn
Where we are

<table>
<thead>
<tr>
<th>Design for understanding</th>
<th>Small scale: One/few objects</th>
<th>Mid scale: Many objects</th>
<th>Large scale: Subsystems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design for understanding</td>
<td>Subtype Polymorphism ✓</td>
<td>Domain Analysis ✓</td>
<td>GUI vs Core ✓</td>
</tr>
<tr>
<td>Design for understanding</td>
<td>Information Hiding, Contracts ✓</td>
<td>Inheritance &amp; Del. ✓</td>
<td>Frameworks and Libraries ✓, APIs ✓</td>
</tr>
<tr>
<td>Design for understanding</td>
<td>Immutability ✓</td>
<td>Responsibility Assignment,</td>
<td>Module systems, microservices ✓</td>
</tr>
<tr>
<td>Design for understanding</td>
<td>Types ✓</td>
<td>Design Patterns, Antipattern ✓</td>
<td>Testing for Robustness ✓</td>
</tr>
<tr>
<td>Design for understanding</td>
<td>Static Analysis ✓</td>
<td>Promises/ Reactive P. ✓</td>
<td>CI ✓, DevOps ✓, Teams</td>
</tr>
<tr>
<td>Design for understanding</td>
<td>Unit Testing ✓</td>
<td>Integration Testing ✓</td>
<td></td>
</tr>
</tbody>
</table>

Small scale: One/few objects
- Subtype Polymorphism ✓
- Information Hiding, Contracts ✓
- Immutability ✓
- Types ✓
- Static Analysis ✓
- Unit Testing ✓

Mid scale: Many objects
- Domain Analysis ✓
- Inheritance & Del. ✓
- Responsibility Assignment, Design Patterns, Antipattern ✓
- Promises/ Reactive P. ✓
- Integration Testing ✓

Large scale: Subsystems
- GUI vs Core ✓
- Frameworks and Libraries ✓, APIs ✓
- Module systems, microservices ✓
- Testing for Robustness ✓
- CI ✓, DevOps ✓, Teams

Design for understanding change/ext. reuse robustness...

Small scale:
- One/few objects

Mid scale:
- Many objects

Large scale:
- Subsystems

Design for understanding change/ext. reuse robustness...
Design Goals

● What are we looking for in design?
  ○ Reuse
  ○ Readability
  ○ Robustness
  ○ Extensibility
  ○ Performance
  ○ ...

...
A simple function
...in sync world

How to make this asynchronous?

- What needs to “happen first”?
- What is the control-flow in callback world?
Next Step: Async/Await

- Async functions return a promise
  - May wrap concrete values
  - May return rejected promises on exceptions
- Allowed to ‘await’ synchronously

```javascript
async function copyAsyncAwait(source: string, dest: string) {
  let statPromise = promisify(fs.stat)

  // Stat dest.
  try {
    await statPromise(dest)
  } catch (_) {
    console.log("Destination already exists")
    return
  }
}
```
The Promise Pattern

- **Problem:** one or more values we will need will arrive later
  - At some point we **must** wait
- **Solution:** an abstraction for *expected values*
- **Consequences:**
  - Declarative behavior for when results become available (*conf.* callbacks)
  - Need to provide paths for normal and abnormal execution
    - E.g., then() and catch()
  - May want to allow combinators
  - Debugging requires some rethinking
Generator Pattern

- Problem: process a collection of indeterminate size
- Solution: provide data points on request when available
- Consequences:
  - Each call to ‘next’ is like awaiting a promise
  - A generator can be infinite, and can announce if it is complete.
  - Generators can be lazy, only producing values on demand
    - Or producing promises
- Where might this be useful?
Traversing a collection

- Since Java 1.0:
  Vector arguments = …;
  for (int i = 0; i < arguments.size(); ++i) {
      System.out.println(arguments.get(i));
  }

- Java 1.5: enhanced for loop
  List<String> arguments = …;
  for (String s : arguments) {
      System.out.println(s);
  }

- Works for every implementation of `Iterable`
  ```java
  public interface Iterable<E> {
      public Iterator<E> iterator();
  }
  public interface Iterator<E> {
      boolean hasNext();
      E next();
      void remove();
  }
  ```

- In JavaScript (ES6)
  ```javascript
  let arguments = …
  for (const s of arguments) {
      console.log(s);
  }
  ```

- Works for every implementation with a “magic” function `[Symbol.iterator]` providing an iterator
  ```javascript
  interface Iterator<T> {
      next(value?: any): IteratorResult<T>;
      return?(value?: any): IteratorResult<T>;
      throw?(e?: any): IteratorResult<T>;
  }
  interface IteratorReturnResult<TReturn> {
      done: true;
      value: TReturn;
  }
  ```
List<String> results = stream.map(Object::toString)
    .filter(s -> pattern.matcher(s).matches())
    .collect(Collectors.toList());

int sum = numbers.parallelStream().reduce(0, Integer::sum);

for (let [odd, even] in numbers.split(n => n % 2, n => !(n % 2)).zip()) {
    console.log(`odd = ${odd}, even = ${even}`); // [1, 2], [3, 4], ...
}

Stream(people).filter({age: 23}).flatMap("children").map("firstName")
    .distinct().filter(/a.*\//i).join(",");
HW5: Santorini with God Cards and GUI
Principles of Software Construction: Objects, Design, and Concurrency

Events Everywhere

Christian Kästner  Vincent Hellendoorn
Where we are

<table>
<thead>
<tr>
<th>Design for understanding change/ext. reuse robustness ...</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Small scale:</strong> One/few objects</td>
</tr>
<tr>
<td>Subtype Polymorphism ✓</td>
</tr>
<tr>
<td>Information Hiding, Contracts ✓</td>
</tr>
<tr>
<td>Immutability ✓</td>
</tr>
<tr>
<td>Types ✓</td>
</tr>
<tr>
<td>Static Analysis ✓</td>
</tr>
<tr>
<td>Unit Testing ✓</td>
</tr>
<tr>
<td><strong>Mid scale:</strong> Many objects</td>
</tr>
<tr>
<td>Domain Analysis ✓</td>
</tr>
<tr>
<td>Inheritance &amp; Del. ✓</td>
</tr>
<tr>
<td>Responsibility Assignment, Design Patterns, Antipattern ✓</td>
</tr>
<tr>
<td>Promises/Reactive P. ✓</td>
</tr>
<tr>
<td>Integration Testing ✓</td>
</tr>
<tr>
<td><strong>Large scale:</strong> Subsystems</td>
</tr>
<tr>
<td>GUI vs Core ✓</td>
</tr>
<tr>
<td>Frameworks and Libraries ✓</td>
</tr>
<tr>
<td>APIs ✓</td>
</tr>
<tr>
<td>Module systems, microservices ✓</td>
</tr>
<tr>
<td>Testing for Robustness ✓</td>
</tr>
<tr>
<td>CI ✓, DevOps ✓</td>
</tr>
<tr>
<td>Teams</td>
</tr>
</tbody>
</table>
class Stack {
    readonly #inner: any[]

    constructor (inner: any[]) {
        this.#inner=inner
    }

    push(o: any): Stack {
        const newInner = this.#inner.slice()
        newInner.push(o)
        return new Stack(newInner)
    }

    peek(): any {
        return this.#inner[this.#inner.length-1]
    }

    getInner(): any[] {
        return this.#inner
    }
}
Useful analogy: Spreadsheets

Cells contain data or formulas

Formula cells are computed automatically whenever input data changes
# Beyond Spreadsheet Cells

<table>
<thead>
<tr>
<th></th>
<th>SINGLE</th>
<th>MULTIPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pull</td>
<td>Function</td>
<td>Iterator</td>
</tr>
<tr>
<td>Push</td>
<td>Promise</td>
<td>Observable</td>
</tr>
</tbody>
</table>

[https://rxjs.dev/guide/observable](https://rxjs.dev/guide/observable)
The Adapter Design Pattern

Client Interface
+ method(data)

Adapter
- adaptee: Service
+ method(data)

Service
+ serviceMethod(specialData)

specialData = convertToServiceFormat(data)
return adaptee.serviceMethod(specialData)

https://refactoring.guru/design-patterns/adapter
Recall: Separating application core and GUI

- Reduce coupling: do not allow core to depend on UI
- Create and test the core without a GUI
  - Use the Observer pattern to communicate information from the core (Model) to the GUI (View)
Client-Server Programming forces Frontend-Backend Separation

Backend (Java/Node): Data, logic, rendering
Frontend (Browser, HTML, JavaScript): Text, buttons

Trick to let backend push information to frontend: Keep http request open, append to page (compare to stream)
Alternative: regular pulling
Principles of Software Construction: Objects, Design, and Concurrency

Libraries and Frameworks
(Design for large-scale reuse)

Christian Kästner  Vincent Hellendoorn
Michael Hilton
Where we are

<table>
<thead>
<tr>
<th>Small scale: One/few objects</th>
<th>Mid scale: Many objects</th>
<th>Large scale: Subsystems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtype Polymorphism ✓</td>
<td>Domain Analysis ✓</td>
<td>GUI vs Core ✓</td>
</tr>
<tr>
<td>Information Hiding, Contracts ✓</td>
<td>Inheritance &amp; Del. ✓</td>
<td>Frameworks and Libraries ✓, APIs ✓</td>
</tr>
<tr>
<td>Immutability ✓</td>
<td>Responsibility Assignment, Design Patterns, Antipattern ✓</td>
<td>Module systems, microservices ✓</td>
</tr>
<tr>
<td>Types ✓</td>
<td>Promises/ Reactive P. ✓</td>
<td>Testing for Robustness ✓</td>
</tr>
<tr>
<td>Static Analysis ✓</td>
<td>Integration Testing ✓</td>
<td>CI ✓, DevOps ✓, Teams</td>
</tr>
<tr>
<td>Unit Testing ✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Design for understanding change/ext.

reuse

robustness

Small scale: One/few objects

Mid scale: Many objects

Large scale: Subsystems

...
Reuse and variation: Flavors of Linux
Terminology: Libraries

- **Library**: A set of classes and methods that provide reusable functionality
Terminology: Frameworks

- Framework: Reusable skeleton code that can be customized into an application

- Framework calls back into client code
  - The Hollywood principle: “Don’t call us. We’ll call you.”
An aside: Plugins could be reusable too…

public class Application extends JFrame implements InputProvider {
    private JTextField textField;
    private Plugin plugin;
    public Application() {
    }
    protected void init(Plugin p) {
        p.setApplication(this);
        this.plugin = p;
    }
    public interface Plugin {
        String getApplicationTitle();
        String getButtonText();
        String getInititalText();
        void buttonClicked();
        void setApplication(InputProvider app);
    }
    public class CalcPlugin implements Plugin {
        private InputProvider app;
        public void setApplication(InputProvider app) {
            this.app = app;
        }
        public String getButtonText() {
            return "calculate";
        }
        public String getInititalText() {
            return "10 / 2 + 6";
        }
        public void buttonClicked() {
            JOptionPane.showMessageDialog(null, "The result of " + application.getInput() + " is " + calculate(application.getInput()));
        }
        public String getApplicationTitle() {
            return "My Great Calculator";
        }
    }
    public interface InputProvider {
        String getInput();
    }
}
The use vs. reuse dilemma

- Large rich components are very useful, but rarely fit a specific need
- Small or extremely generic components often fit a specific need, but provide little benefit

“maximizing reuse minimizes use”

C. Szyperski
Principles of Software Construction

API Design

Christian Kästner  Vincent Hellendoorn
(Many slides originally from Josh Bloch)
## Where we are

<table>
<thead>
<tr>
<th>Design for understanding change/ext. reuse robustness ...</th>
</tr>
</thead>
</table>

### Small scale: One/few objects
- Subtype
  - Polymorphism ✓
- Information Hiding, Contracts ✓
- Immutability ✓
- Types ✓
- Static Analysis ✓
- Unit Testing ✓

### Mid scale: Many objects
- Domain Analysis ✓
- Inheritance & Del. ✓
- Responsibility Assignment, Design Patterns, Antipattern ✓
- Promises/Reactive P. ✓
- Integration Testing ✓

### Large scale: Subsystems
- GUI vs Core ✓
- Frameworks and Libraries ✓, **APIs** ✓
- Module systems, microservices ✓
- Testing for Robustness ✓
- CI ✓, DevOps ✓, Teams
Composing Templates

(Corresponds to Fragments in Handlebars)

Nest templates

Pass arguments (properties) between templates

Try it:
https://reactjs.org/redirect-to-codepen/components-and-props/composing-components
Public APIs are forever

- Eclipse (IBM)
- JDT Plugin (IBM)
- CDT Plugin (IBM)
- UML Plugin (third party)
- Somebody on the web
- Somebody on the web
- Somebody on the web
- Somebody on the web
- Somebody on the web
- Somebody on the web
- Somebody on the web
- third party plugin
Hyrum’s Law

“With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody.”

https://www.hyrumslaw.com/

https://xkcd.com/1172/
The process of API design – 1-slide version

Not sequential; if you discover shortcomings, iterate!

1. **Gather requirements** skeptically, including **use cases**
2. **Choose an abstraction** (model) that appears to address use cases
3. **Compose a short API sketch** for abstraction
4. **Apply API sketch to use cases** to see if it works
   - If not, go back to step 3, 2, or even 1
5. **Show API** to anyone who will look at it
6. **Write prototype** implementation of API
7. **Flesh out** the documentation & harden implementation
8. **Keep refining it** as long as you can
Applying Information hiding: Factories

```java
public class Rectangle {
    public Rectangle(Point e, Point f) ...
}

// ...
Point p1 = PointFactory.Construct(...);
// new PolarPoint(...); inside
Point p2 = PointFactory.Construct(...);
// new PolarPoint(...); inside
Rectangle r = new Rectangle(p1, p2);
```
Aside: The *Factory Method* Design Pattern

From: [https://refactoring.guru/design-patterns/factory-method](https://refactoring.guru/design-patterns/factory-method)
Boilerplate Code

import org.w3c.dom.*;
import java.io.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;

/** DOM code to write an XML document to a specified output stream. */
static final void writeDoc(Document doc, OutputStream out) throws IOException {
    try {
        Transformer t = TransformerFactory.newInstance().newTransformer();
        t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, doc.getDoctype().getSystemId());
        t.transform(new DOMSource(doc), new StreamResult(out)); // Does actual writing
    } catch (TransformerException e) {
        throw new AssertionError(e); // Can’t happen!
    }
}
Principles of Software Construction

API Design (Part 2)

Christian Kästner    Vincent Hellendoorn
(With slides from Josh Bloch)
Where we are

<table>
<thead>
<tr>
<th>Small scale: One/few objects</th>
<th>Mid scale: Many objects</th>
<th>Large scale: Subsystems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtype</td>
<td>Domain Analysis ✓</td>
<td>GUI vs Core ✓</td>
</tr>
<tr>
<td>Polymorphism ✓</td>
<td>Inheritance &amp; Del. ✓</td>
<td>Frameworks and Libraries ✓, <strong>APIs ✓</strong></td>
</tr>
<tr>
<td>Information Hiding, Contracts ✓</td>
<td>Responsibility Assignment, Design Patterns, Antipattern ✓</td>
<td>Module systems, microservices ✓</td>
</tr>
<tr>
<td>Immutability ✓</td>
<td>Types ✓</td>
<td>Testing for Robustness ✓</td>
</tr>
<tr>
<td>Types ✓</td>
<td>Static Analysis ✓</td>
<td>CI ✓, DevOps ✓</td>
</tr>
<tr>
<td>Types ✓</td>
<td>Unit Testing ✓</td>
<td><strong>Teams</strong></td>
</tr>
<tr>
<td>Static Analysis ✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unit Testing ✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integration Testing ✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Design for
understanding
change/ext.
reuse
robustness
...

Small scale: One/few objects
- Subtype
- Polymorphism ✓
- Information Hiding, Contracts ✓
- Immutability ✓
- Types ✓
- Static Analysis ✓
- Unit Testing ✓

Mid scale: Many objects
- Domain Analysis ✓
- Inheritance & Del. ✓
- Responsibility Assignment, Design Patterns, Antipattern ✓
- Types ✓
- Static Analysis ✓
- Unit Testing ✓
- Integration Testing ✓

Large scale: Subsystems
- GUI vs Core ✓
- Frameworks and Libraries ✓, **APIs ✓**
- Module systems, microservices ✓
- Testing for Robustness ✓
- CI ✓, DevOps ✓, **Teams**
Teamwork

Teamwork essential in software projects

Teamwork needed to scale available work and available skills

Teamwork is a key motivation for design for understandability, documentation, etc
Good names drive good design

● Be consistent
  ○ `computeX()` vs. `generateX()`?
  ○ `deleteX()` vs. `removeX()`?

● Avoid cryptic abbreviations
  ○ Good: `Font`, `Set`, `PrivateKey`, `Lock`, `ThreadFactory`, `TimeUnit`, `Future<T>`
  ○ Bad: `DynAnyFactoryOperations`, `_BindingIteratorImplBase`, `ENCODING_CDR_ENCAPS`, `OMGVMCID`
Principle: Favor composition over inheritance

// A Properties instance maps Strings to Strings
public class Properties extends HashTable {
    public Object put(Object key, Object value);
    ...
}

public class Properties {
    private final HashTable data = new HashTable();
    public String put(String key, String value) {
        data.put(key, value);
    }
    ...
}
Principle: Fail fast

- Report errors as soon as they are detectable
  - Check preconditions at the beginning of each method
  - Avoid dynamic type casts, run-time type-checking

```java
// A Properties instance maps Strings to Strings
public class Properties extends HashTable {
    public Object put(Object key, Object value);

    // Throws ClassCastException if this instance
    // contains any keys or values that are not Strings
    public void save(OutputStream out, String comments);
}
```
CRUD Operations

Path correspond to nouns, not verbs, nesting common:

- `/articles`, `/state`, `/game`
  `/articles/:id/comments`

GET (receive), POST (submit new), PUT (update), and DELETE requests sent to those paths

Parameters for filtering, searching, sorting, e.g., `/articles?sort=date`

```javascript
const express = require('express);
const bodyParser = require('body-parser');
const app = express();
app.use(bodyParser.json()); // JSON input
app.get('/articles', (req, res) => {
    const articles = [];
    // code to retrieve an article...
    res.json(articles);
});
apppost('/articles', (req, res) => {
    // code to add a new article...
    res.json(req.body);
});
appput('/articles/:id', (req, res) => {
    const { id } = req.params;
    // code to update an article...
    res.json(req.body);
});
app.delete('/articles/:id', (req, res) => {
    const { id } = req.params;
    // code to delete an article...
    res.json({ deleted: id });
});
app.listen(3000, () => console.log('server started'));
```
Announcements
Documentation
Migration guide
Easy and fast to publish and use for developers

Breaking changes easy
More common to remove technical debt, fix APIs
Signaling intention with SemVer
No central release planning
Parallel releases more common

Upstream
HW6: Data Analytics Framework
Principles of Software Construction: Objects, Design, and Concurrency

Design for Robustness: Distributed Systems

Christian Kästner

Vincent Hellendoorn
Where we are

**Design for understanding**
change/ext. reuse robustness ...

**Small scale:**
One/few objects
- Subtype Polymorphism ✓
- Information Hiding, Contracts ✓
- Immutability ✓
- Types ✓
- Static Analysis ✓
- Unit Testing ✓

**Mid scale:**
Many objects
- Domain Analysis ✓
- Inheritance & Del. ✓
- Responsibility Assignment, Design Patterns, Antipattern ✓
- Promises/ Reactive P. ✓
- Integration Testing ✓

**Large scale:**
Subsystems
- GUI vs Core ✓
- Frameworks and Libraries ✓, APIs ✓
- Module systems, microservices ✓
- Testing for Robustness ✓
- CI ✓, DevOps ✓, Teams
Retry!

- Still need an exit-strategy
  - Learn **HTTP response codes**
    - Don’t bother retrying on a 403 (go find out why)
  - Use the API response, if any

---

```javascript
const delay = retryCount => new Promise(resolve =>
  setTimeout(resolve, 10 ** retryCount));

const getResource = async (retryCount = 0, lastError = null) => {
  if (retryCount > 5) throw new Error(lastError);
  try {
    return apiCall();
  } catch (e) {
    await delay(retryCount);
    return getResource(retryCount + 1, e);
  }
```


---
Proxy Design Pattern

- Local representative for remote object
  - Create expensive obj on-demand
  - Control access to an object
- Hides extra “work” from client
  - Add extra error handling, caching
  - Uses indirection
Ever looked at NPM Install’s output?
Eliminating Android dependency

@Test void testGetFriends() {
    assert getFriends() == ...;
}

List<Friend> getFriends() {
    Connection c = http.getConnection();
    FacebookAPI api = new FacebookAPI(c);
    List<Node> persons = api.getFriends("john");
    for (Node person1 : persons) {
        ...
    }
    return result;
}
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
Principle: Modular Protection

- Errors should be contained and isolated
  - A failing printer should not corrupt a document
  - Handle exceptions locally as much as possible, return useful feedback
  - Don’t do this:

```
HTTP Status 500 -

Type: Exception report

Message: The server encountered an internal error that prevented it from fulfilling this request.

Exception:
java.lang.NullPointerException
  nl.hu.sp.lesson1.dynamicexample.LogoutServlet.doGet (LogoutServlet.java:39)
  javax.servlet.http.HttpServlet.service (HttpServlet.java:618)
  javax.servlet.http.HttpServlet.service (HttpServlet.java:725)
  org.apache.tomcat.websocket.server.WsFilter.doFilter (WsFilter.java:52)

Note: The full stack trace of the root cause is available in the Apache Tomcat/8.0.5 logs.
```

Apache Tomcat/8.0.5
Ensuring Idempotence

- How about writing/sending new data?
  - Could fail anywhere
    - Including in displaying success message after payment!
  - POST is not idempotent
  - Use Unique Identifiers
  - Server keeps track of requests already handled

```bash
curl https://api.stripe.com/v1/charges \\
  -u sk_test_BQokikJ0vBiiI2HlWgH4olfQ2: \\
  -H "Idempotency-Key: AGJ6FJMKGQIphUTX" \\
  -d amount=2000 \\
  -d currency=usd \\
  -d description="Charge for Brandur" \\
  -d customer=cus_A8Z5MHwQS7jUmZ
```

https://stripe.com/blog/idempotency
Principles of Software Construction: Objects, Design, and Concurrency

Organizing Systems at Scale: Modules, Services, Architectures

Christian Kästner  Vincent Hellendoorn
## Where we are

<table>
<thead>
<tr>
<th>Design for understanding change/ext. reuse robustness ...</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Small scale:</strong> One/few objects</td>
</tr>
<tr>
<td>Subtype</td>
</tr>
<tr>
<td>Polymorphism ✓</td>
</tr>
<tr>
<td>Information Hiding, Contracts ✓</td>
</tr>
<tr>
<td>Immutability ✓</td>
</tr>
<tr>
<td>Types</td>
</tr>
<tr>
<td>Unit Testing ✓</td>
</tr>
<tr>
<td><strong>Mid scale:</strong> Many objects</td>
</tr>
<tr>
<td>Domain Analysis ✓</td>
</tr>
<tr>
<td>Inheritance &amp; Del. ✓</td>
</tr>
<tr>
<td>Responsibility Assignment, Design Patterns, Antipattern ✓</td>
</tr>
<tr>
<td>Promises/ Reactive P. ✓</td>
</tr>
<tr>
<td>Integration Testing ✓</td>
</tr>
<tr>
<td><strong>Large scale:</strong> Subsystems</td>
</tr>
<tr>
<td>GUI vs Core ✓</td>
</tr>
<tr>
<td>Frameworks and Libraries ✓</td>
</tr>
<tr>
<td>APIs ✓</td>
</tr>
<tr>
<td><strong>Module systems, microservices</strong></td>
</tr>
<tr>
<td>Testing for Robustness ✓</td>
</tr>
<tr>
<td>CI ✓, DevOps, Teams</td>
</tr>
</tbody>
</table>

*Table summary:

- Small scale: One/few objects
  - Subtype
  - Polymorphism ✓
  - Information Hiding, Contracts ✓
  - Immutability ✓
  - Types
  - Unit Testing ✓

- Mid scale: Many objects
  - Domain Analysis ✓
  - Inheritance & Del. ✓
  - Responsibility Assignment, Design Patterns, Antipattern ✓
  - Promises/ Reactive P. ✓
  - Integration Testing ✓

- Large scale: Subsystems
  - GUI vs Core ✓
  - Frameworks and Libraries ✓, APIs ✓
  - **Module systems, microservices**
  - Testing for Robustness ✓
  - CI ✓, DevOps, Teams
Packages enough?

edu.cmu.cs214.santorini
edu.cmu.cs214.santorini.gui
edu.cmu.cs214.santorini.godcards
edu.cmu.cs214.santorini.godcards.impl
edu.cmu.cs214.santorini.logic
edu.cmu.cs214.santorini.utils
ES2015 Modules

Syntax extension for modules (instead of module pattern)

Explicit imports / exports

Static import names (like Java), supports better reasoning by tools

```javascript
import { Location } from './location'
import { Game } from './game'
import { Board } from './board'
// module code
export { Worker, newWorker }
```
The Diamond Problem

What now?
Recommended reading:
Microservices

Microservices Everywhere
Handle Errors Locally

Service encapsulation hides failure Service E behind Service B such that it is not observable by Service A. (execution either the same as Service B, C success and D failure combo or Service C success and B and D failure combo, depending on B.)
Apache Kafka

Principles of Software Construction: Objects, Design, and Concurrency

A Quick Tour of all 23 GoF Design Patterns

Christian Kästner  Vincent Hellendoorn
Where we are

<table>
<thead>
<tr>
<th>Small scale: One/few objects</th>
<th>Mid scale: Many objects</th>
<th>Large scale: Subsystems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtype Polymorphism ✓</td>
<td>Domain Analysis ✓</td>
<td>GUI vs Core ✓</td>
</tr>
<tr>
<td>Information Hiding, Contracts ✓</td>
<td>Inheritance &amp; Del. ✓</td>
<td>Frameworks and Libraries ✓, APIs ✓</td>
</tr>
<tr>
<td>Immutability ✓</td>
<td>Responsibility Assignment, Design Patterns, Antipattern ✓</td>
<td>Module systems, microservices ✓</td>
</tr>
<tr>
<td>Types</td>
<td>Promises/ Reactive P. ✓</td>
<td>Testing for Robustness ✓</td>
</tr>
<tr>
<td>Unit Testing ✓</td>
<td>Integration Testing ✓</td>
<td>CI ✓, DevOps, Teams</td>
</tr>
</tbody>
</table>

Design for
understanding
change/ext.
reuse
robustness
...

Small scale:
- One/few objects

Mid scale:
- Many objects
  - Domain Analysis ✓
  - Inheritance & Del. ✓
  - Responsibility Assignment, Design Patterns, Antipattern ✓
  - Promises/ Reactive P. ✓
  - Integration Testing ✓

Large scale:
- Subsystems
  - GUI vs Core ✓
  - Frameworks and Libraries ✓, APIs ✓
  - Module systems, microservices ✓
  - Testing for Robustness ✓
  - CI ✓, DevOps, Teams
1. Creational Patterns

1. Abstract factory
2. Builder
3. Factory method
4. Prototype
5. Singleton
Singleton Illustration

```java
public class Elvis {
    private static final Elvis ELVIS = new Elvis();
    public static Elvis getInstance() { return ELVIS; }
    private Elvis() {}
    ...
}
```

```javascript
const elvis = { ... }
function getElvis() {
    export { getElvis }
}```
II. Structural Patterns

1. Adapter
2. Bridge
3. Composite
4. Decorator
5. Façade
6. Flyweight
7. Proxy
Decorator vs Strategy?

```java
interface GameLogic {
    isValidMove(w, x, y)
    move(w, x, y)
}

class BasicGameLogic implements GameLogic {
    constructor(board) { ... }
    isValidMove(w, x, y) { ... }
    move(w, x, y) { ... }
}

class AbstractGodCardDecorator implements GameLogic {
    move(w, x, y) { /* super.move(w, x, y) + checkWinner */ }
}

class PanDecorator extends AbstractGodCardDecorator implements GameLogic {
    move(w, x, y) { /* super.move(w, x, y) + checkWinner */ }
}

// Example usage
```

III. Behavioral Patterns

1. Chain of Responsibility
2. Command
3. Interpreter
4. Iterator
5. Mediator
6. Memento
7. Observer
8. State
9. Strategy
10. Template method
11. Visitor
Strategy vs Template Method

```
AbstractClass
...
+ templateMethod()
 + step1()
 + step2()
 + step3()
 + step4()

ConcreteClass1
...
+ step3()
+ step4()

ConcreteClass2
...
+ step1()
+ step2()
+ step3()
+ step4()
```

```
Context
- strategy
 + setStrategy(strategy)
 + doSomething()

Strategy
+ execute(data)

ConcreteStrategies
+ execute(data)

Client
strategy.execute()
```
Principles of Software Construction: Objects, Design, and Concurrency

{Static & Dynamic} x {Typing & Analysis}

Christian Kästner  Vincent Hellendoorn
Where we are

<table>
<thead>
<tr>
<th>Small scale: One/few objects</th>
<th>Mid scale: Many objects</th>
<th>Large scale: Subsystems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtype</td>
<td>Domain Analysis ✓</td>
<td>GUI vs Core ✓</td>
</tr>
<tr>
<td>Polymorphism ✓</td>
<td>Inheritance &amp; Del. ✓</td>
<td>Frameworks and Libraries ✓, APIs ✓</td>
</tr>
<tr>
<td>Information Hiding, Contracts ✓</td>
<td>Responsibility Assignment, Design Patterns, Antipattern ✓</td>
<td>Module systems, microservices ✓</td>
</tr>
<tr>
<td>Immutability ✓</td>
<td>Promises/Reactive P. ✓</td>
<td>Testing for Robustness ✓</td>
</tr>
<tr>
<td>Types ✓</td>
<td>Integration Testing ✓</td>
<td>CI ✓, DevOps ✓, Teams</td>
</tr>
<tr>
<td>Static Analysis ✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unit Testing ✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Design for understanding
change/ext.
reuse
robustness
...

Small scale:
- One/few objects

Mid scale:
- Many objects

Large scale:
- Subsystems

...
How Do You Find Bugs?

- Run it?

```java
public class Fails {
    public static void main(String[] args) {
        getValue(i: null);
    }

    private static int getValue(Integer i) {
        return i.intValue();
    }
}
```

Exception in thread "main" java.lang.NullPointerException
    at misc.Fails.getValue(Fails.java:9)
    at misc.Fails.main(Fails.java:5)
Static vs. Dynamic Typing

● The more knowledge we inject in the code, the more bugs we can catch at compile time
  ○ Types, nullity annotations, invariants

● Is it worth it?
  ○ Dynamic typing can severely limit inference
  ○ But… static types are a lot of work
Static Analysis

- How?
  - Program analysis + Vocabulary of patterns

https://deepsorce.io/blog/introduction-static-code-analysis/
Static Analysis

● **Step 3: register analyzers**
  ○ At the core: walk the tree
  ○ Sometimes more complex

```python
class UnusedImportChecker(BaseChecker):
    def __init__(self):
        self.import_map = defaultdict(set)
        self.name_map = defaultdict(set)

    def _add_imports(self, self, node):
        for import_name in node.names:
            # Store only top-level module name ("os.path" -> "os").
            # We can't easily detect when "os.path" is used.
            name = import_name.name.partition(".")[0]
            self.import_map[self.filename].add((name, node.lineno))

    def visit_Import(self, self, node):
        self._add_imports(node)

    def visit_ImportFrom(self, self, node):
        self._add_imports(node)

    def visit_Name(self, self, node):
        # We only add those nodes for which a value is being read from.
        if isinstance(node.ctx, ast.Load):
            self.name_map[self.filename].add(node.id)
```
Static Analysis at Google

- Centered around FindBugs (succeeded by SpotBugs)
  - Essentially, a huge collection of risky patterns on Java bytecode
  - Annotated with five levels of concern
TriCoder

```java
package com.google.devtools.staticanalysis;

public class Test {
    public boolean foo() {
        return getString() == "foo".toString();
    }
}
```

- **Lint**
  - Missing a Javadoc comment.
  - Please fix

- **ErrorProne**
  - String comparison using reference equality instead of value equality
  - Please fix
Principles of Software Construction: Objects, Design, and Concurrency

DevOps

Christian Kästner  Vincent Hellendoorn
Where we are

<table>
<thead>
<tr>
<th>Small scale: One/few objects</th>
<th>Mid scale: Many objects</th>
<th>Large scale: Subsystems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtype</td>
<td>Domain Analysis ✓</td>
<td>GUI vs Core ✓</td>
</tr>
<tr>
<td>Polymorphism ✓</td>
<td>Inheritance &amp; Del. ✓</td>
<td>Frameworks and Libraries ✓, APIs ✓</td>
</tr>
<tr>
<td>Information Hiding, Contracts ✓</td>
<td>Responsibility Assignment, Design Patterns, Antipattern ✓</td>
<td>Module systems, microservices ✓</td>
</tr>
<tr>
<td>Immutability ✓</td>
<td>Promises/Reactive P. ✓</td>
<td>Testing for Robustness ✓</td>
</tr>
<tr>
<td>Types ✓</td>
<td>Integration Testing ✓</td>
<td>CI ✓, DevOps, Teams</td>
</tr>
<tr>
<td>Static Analysis ✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unit Testing ✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Design for understanding change/ext. reuse robustness...

Small scale:
- One/few objects

Mid scale:
- Many objects

Large scale:
- Subsystems

- GUI vs Core ✓
- Frameworks and Libraries ✓, APIs ✓
- Module systems, microservices ✓
- Testing for Robustness ✓
- CI ✓, DevOps, Teams
Release management with branches
Heavy Tooling and Automation
A/B Testing

**Original: 2.3%**

**Long Form: 4.3%**
Looking Forward: Beyond Code-Level Concerns
### Where we are

**Small scale:** One/few objects
- Subtype
- Polymorphism ✓
- Information Hiding, Contracts ✓
- Immutability ✓
- Types ✓
- Static Analysis ✓
- Unit Testing ✓

**Mid scale:** Many objects
- Domain Analysis ✓
- Inheritance & Del. ✓
- Responsibility Assignment, Design Patterns, Antipattern ✓
- Promises/Reactive P. ✓
- Integration Testing ✓

**Large scale:** Subsystems
- GUI vs Core ✓
- Frameworks and Libraries ✓, APIs ✓
- Module systems, microservices ✓
- Testing for Robustness ✓
- CI ✓, DevOps ✓, Teams

---

**Design for understanding**, change/ext., reuse, robustness...

<table>
<thead>
<tr>
<th>Small scale: One/few objects</th>
<th>Mid scale: Many objects</th>
<th>Large scale: Subsystems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtype</td>
<td>Domain Analysis ✓</td>
<td>GUI vs Core ✓</td>
</tr>
<tr>
<td>Polymorphism ✓</td>
<td>Inheritance &amp; Del. ✓</td>
<td>Frameworks and Libraries ✓, APIs ✓</td>
</tr>
<tr>
<td>Information Hiding, Contracts ✓</td>
<td>Responsibility Assignment, Design Patterns, Antipattern ✓</td>
<td>Module systems, microservices ✓</td>
</tr>
<tr>
<td>Immutability ✓</td>
<td>Promises/Reactive P. ✓</td>
<td>Testing for Robustness ✓</td>
</tr>
<tr>
<td>Types ✓</td>
<td>Integration Testing ✓</td>
<td>CI ✓, DevOps ✓, Teams</td>
</tr>
<tr>
<td>Static Analysis ✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unit Testing ✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Small scale
  - One/few objects
  - Design for understanding, change/ext., reuse, robustness...

- Mid scale
  - Many objects
  - Design for understanding, change/ext., reuse, robustness...

- Large scale
  - Subsystems
  - Design for understanding, change/ext., reuse, robustness...
This Course

We focused on code-level concerns

Writing maintainable, extensible, robust, and correct code

Design from classes to subsystems

Testing, concurrency, basic user interfaces
From Programming to Software Engineering
“Software Engineering” was a provocative term
Compare to other forms of engineering

- e.g., Producing a car or bridge
  - Estimable costs and risks
  - Well-defined expected results
  - High quality
- Separation between plan and production
- Simulation before construction
- Quality assurance through measurement
- Potential for automation
Software engineering in the real world

- e.g., HealthCare.gov
  - Estimable costs and risks
  - Well-defined expected results
  - High quality
- Separation between plan and production
- Simulation before construction
- Quality assurance through measurement
- Potential for automation
Software is written by humans

*Sociotechnical system*: interlinked system of people, technology, and their environment

Key challenges in how to

- identify what to build (requirements)
- coordinate people building it (process)
- assure quality (speed, safety, fairness)
- contain risk, time and budget (management)
- sustain a community (open source, economics)
Process
Example: Process

Win Royce and Barry Boehm, 1970
Phase That a Defect Is Created

Cost to Correct

Phase That a Defect Is Corrected

Requirements Architecture Detailed design Construction Maintenance

Requirements Architecture Detailed design Construction Maintenance

Agile in a nutshell

● A project management approach that seeks to respond to change and unpredictability, primarily using incremental, iterative work sequences (often called “sprints”).
● Also: a collection of practices to facility that approach.
● All predicated on the principles outlined in “The Manifesto for Agile Software Development.”
The Manifesto for Agile Software Development (2001)

**Value**

- Individuals and interactions *over* Processes and tools
- Working software *over* Comprehensive documentation
- Customer collaboration *over* Contract negotiation
- Responding to change *over* Following a plan
Pair Programming

Driver

Navigator
Scrum Process
QA and Process
Beyond testing
Many QA approaches
Code review, static analysis, formal verification, …

Which to use when, how much?
Twitter is over capacity.
Too many tweets! Please wait a moment and try again.
Lotus Notes

Do you want to send this notice with these comments?

Choose Yes to send as is.
Choose No to send without comments.
Choose Cancel to continue editing.

Yes  No  Cancel
How to get students to write tests?
“We had initially scheduled time to write tests for both front and back end systems, although this never happened.”
“Due to the lack of time, we could only conduct individual pages’ unit testing. Limited testing was done using use cases. Our team felt that this testing process was rushed and more time and effort should be allocated.”
Time estimates (in hours):

<table>
<thead>
<tr>
<th>Activity</th>
<th>Estimated</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>testing plans</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>unit testing</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>validation testing</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>test data</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
How to get students to write tests?

How to get them to take testing seriously, not just as an afterthought?
How to get developers to write tests?
Test Driven Development

- Tests first!
- Popular agile technique
- Write tests as specifications before code
- Never write code without a failing test
- Claims:
  - Design approach toward testable design
  - Think about interfaces first
  - Avoid writing unneeded code
  - Higher product quality (e.g. better code, less defects)
  - Higher test suite quality
  - Higher overall productivity

(CC BY-SA 3.0) Excirial
How to get developers to run tests?
wyvernlang / wyvern

SimpleWyvern-devel Asserting false (works on Linux, so its OK).
# 17 passed
Commit fd7be1c
Compare 0e2a1f..fd7b
ran for 16 sec
3 days ago

This job ran on our legacy infrastructure. Please read our docs on how to upgrade
How to get developers to use static analysis?
https://help.github.com/articles/using-pull-requests/
How to get developers to use static analysis?

```java
package com.google.devtools.staticanalysis;

public class Test {
    public boolean foo() {
        return getString() == "foo".toString();
    }

    public String getString() {
        return new String("foo");
    }
}
```

Lint

Missing a Javadoc comment.

Please fix

Not useful

ErrorProne


Please fix

Suggested fix attached: show

Not useful
Are code reviews worth it?
Requirements
Requirements

- What does the customer want?
- What is required, desired, not necessary? Legal, policy constraints?
- Customers often do not know what they really want; vague, biased by what they see; change their mind; get new ideas...
- Difficult to define requirements precisely
- (Are we building the right thing? Not: Are we building the thing right?)
Lufthansa Flight 2904

- The Airbus A320-200 airplane has a software-based braking system.
- Engaging reverse thrusters while in the air is very dangerous: Only allow breaking when on the ground.
Lufthansa Flight 2904

Two conditions needed to “be on the ground”:

1. Both shock absorber bear a load of 6300 kgs
2. Both wheels turn at 72 knots (83 mph) or faster
Requirements Document

- Call for tenders, proposal evaluation
- Project contract
- Project workplan
- Follow-up directives
- Software architecture
- Software evolution directives
- Software documentation

- Project estimations (size, cost, schedules)
- Software prototype, mockup
- Acceptance test data
- Quality Assurance checklists
- Implementation directives
- User manual
Interviews
Abby Jones

You can edit anything in blue print

- 28 years old
- Employed as an Accountant
- Lives in Cardiff, Wales

Abby has always liked music. When she is on her way to work in the morning, she listens to music that spans a wide variety of styles. But when she arrives at work, she turns it off, and begins her day by scanning all her emails first to get an overall picture before answering any of them. (This extra pass takes time but seems worth it.) Some nights she exercises or stretches, and sometimes she likes to play computer puzzle games like Sudoku.

Background and skills

Abby works as an accountant. She is comfortable with the technologies she uses regularly, but she just moved to this employer 1 week ago, and their software systems are new to her.

Abby says she's a "numbers person"; but she has never taken any computer programming or IT systems spreadsheet classes. She likes Math and knows how to think with numbers. She writes and edits formulas in her work.

In her free time, she also enjoys working with numbers and logic. She especially likes working out puzzles and puzzle games, either on paper or on the computer.

Motivations and Attitudes

Motivations: Abby uses technologies to accomplish her tasks. She learns new technologies if and when she needs to, but prefers to use methods she is already familiar and comfortable with, to keep her focus on the tasks she cares about.

Computer Self-Efficacy: Abby has low confidence about doing unfamiliar computing tasks. If problems arise with her technology, she often blames herself for these problems. This affects whether and how she will persevere with a task if technology problems have arisen.

Attitude toward Risk: Abby's life is a little complicated and she rarely has spare time. So she is risk averse about using unfamiliar technologies that might need her to spend extra time on them, even if the new features might be relevant. She instead performs tasks using familiar features, because they're more predictable about what she will get from them and how much time they will take.

How Abby Works with Information and learns:

Information Processing Style: Abby tends towards a comprehensive and analytical style of processing. She prefers to absorb information through reading and reflection, rather than through active participation or hands-on activities.

Learning: by Process vs. by Tinkering: When learning new technology, Abby prefers to use a step-by-step process. She likes to work through tutorials and documentation, rather than experimenting and trying things out on her own. This helps her feel more confident in her abilities and reduces the risk of making mistakes.

Abby is comfortable with the technologies she uses regularly, but she just moved to this employer 1 week ago, and their software systems are new to her.

Abby says she's a "numbers person"; but she has never taken any computer programming or IT systems spreadsheet classes. She likes Math and knows how to think with numbers. She writes and edits formulas in her work.

In her free time, she also enjoys working with numbers and logic. She especially likes working out puzzles and puzzle games, either on paper or on the computer.

Motivations: Abby uses technologies to accomplish her tasks. She learns new technologies if and when she needs to, but prefers to use methods she is already familiar and comfortable with, to keep her focus on the tasks she cares about.

Computer Self-Efficacy: Abby has low confidence about doing unfamiliar computing tasks. If problems arise with her technology, she often blames herself for these problems. This affects whether and how she will persevere with a task if technology problems have arisen.

Attitude toward Risk: Abby's life is a little complicated and she rarely has spare time. So she is risk averse about using unfamiliar technologies that might need her to spend extra time on them, even if the new features might be relevant. She instead performs tasks using familiar features, because they're more predictable about what she will get from them and how much time they will take.
Advertisement: SE @ CMU

Many courses

Spring: SE for Startups, ML in Production, Program Analysis, WebApps
Fall: Foundations of SE, API Design

Master level: Formal methods, Requirements, Architecture, Agile, QA, DevOps,
Software Project Mgmt, Scalable Systems, Embedded Sys., ...
Technical foundations: ML, Distributed Systems

Many research opportunities -- contact us for pointers

https://www.cmu.edu/scs/isr/reuse/
https://se-phd.isri.cmu.edu/

Software Engineering Concentration / Minor
One Last Survey

Summary

Looking back at one semester of code-level design, testing, and concurrency

Looking forward to human aspects of software engineering, including process and requirements