

Principles of Software Construction: Objects, Design, and Concurrency

Inheritance and delegation

Christian Kästner Vincent Hellendoorn



Notes on HW2

- Common over-testing:
 - MostMistakesFirstOrganizer: not stable
 - InMemoryCardOrganizer: getAllCards order not guaranteed
 - A lot of tests relied on the cards-file → CardLoader loop to get FlashCards
- Common under-testing:
 - FlashCard: reference answer trimming
 - Repeating: more observations than limit
 - Non-repeating organizer: test for a single ‘false’ answer

Quiz

<https://rb.gy/ffljay>

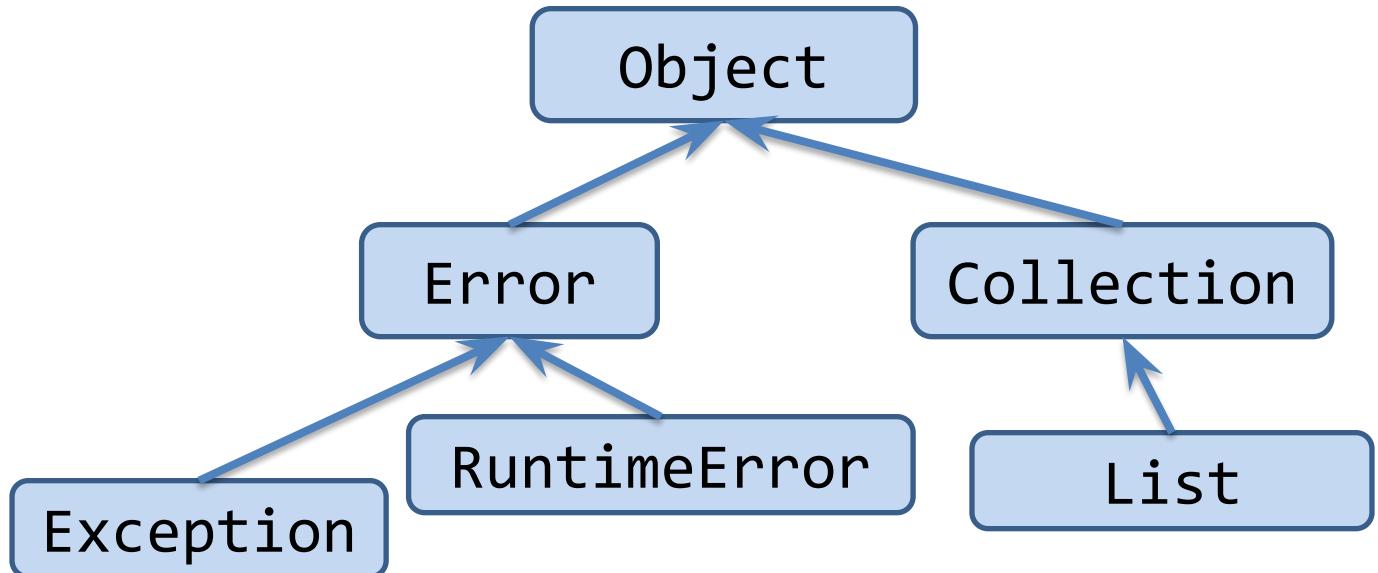


Today

- Class Hierarchies
- Behavioral Subtyping
- Design Goals
 - Template Method Pattern
 - Reuse; relation to coupling
 - When to use inheritance, delegation
- A bit on refactoring

Class Hierarchy

In Java:



Class Hierarchy

Some terminology:

- A class hierarchy is a tree
 - Parent/child relation is called: superclass/subclass
 - A class **extends** its superclass
 - The root is “Object” -- if a class extends nothing explicitly, it extends that
- Primitive types are not in the class hierarchy

Chime In

What does it mean to “extend” a class?

Inheritance enables Extension

```
class Animal {  
    final String name;  
  
    public Animal(String name) {  
        this.name = name;  
    }  
  
    public String identify() {  
        return this.name;  
    }  
}
```

```
class Dog extends Animal {  
    public Dog() {  
        super("dog");  
    }  
  
    Animal animal = new Dog();  
    animal.identify(); // "dog"
```

Inheritance enables Extension

```
class Animal {  
    final String name;  
  
    public Animal(String name) {  
        this.name = name;  
    }  
  
    public String identify() {  
        return this.name;  
    }  
}
```

```
class Dog extends Animal {  
    public Dog() {  
        super("dog");  
    }  
}
```

```
Animal animal = new Dog();  
animal.identify(); // "dog"
```

Declared Type

Compile-time
Check (Java)

Instantiated Type

Is this Allowed?

```
class Animal {  
    final String name;  
  
    public Animal(String name) {  
        this.name = name;  
    }  
  
    public String identify() {  
        return this.name;  
    }  
}
```

```
class Dog extends Animal {  
    public Dog() {  
        super("dog");  
    }  
  
    public String bark() {  
        return "Woof!";  
    }  
}  
  
Dog dog = new Dog();  
dog.bark(); // ??
```

```
Animal animal = new Dog();  
animal.bark(); // ??
```

Is this Allowed?

```
class Animal {  
    final String name;  
  
    public Animal(String name) {  
        this.name = name;  
    }  
  
    public String identify() {  
        return this.name;  
    }  
}
```

```
class Dog extends Animal {  
    public Dog() {  
        super("dog");  
    }  
  
    public Animal identify() {  
        return this;  
    }  
}  
  
Animal animal = new Dog();  
animal.identify(); // ??
```

Behavioral Subtyping

- Formalizes notion of extension

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

Behavioral Subtyping

- Formalizes notion of extension

```
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

Behavioral Subtyping

```
class Animal {  
    final String name;  
  
    public Animal(String name) {  
        this.name = name;  
    }  
  
    public String identify() {  
        return this.name;  
    }  
}
```

```
class Dog extends Animal {  
    public Dog() {  
        super("dog");  
    }  
  
    public String bark() {  
        return "Woof!";  
    }  
  
Dog dog = new Dog();  
dog.bark(); // "Woof"  
  
Animal animal = new Dog();  
animal.bark(); // No such method
```

Behavioral Subtyping

- Subtypes inherit attributes, behavior from their parents
- Subtypes can add new behavior, properties

Is this behavioral subtyping?

```
class Animal {  
  
    final String name;  
  
    public Animal(String name) {  
        this.name = name;  
    }  
  
    public Animal me() {  
        return this;  
    }  
}
```

```
class Dog extends Animal {  
  
    public Dog() {  
        super("dog");  
    }  
  
    public Dog me() {  
        return this;  
    }  
}
```

Is this behavioral subtyping?

```
class Number {  
    value: number;  
  
    constructor(value: number) {  
        this.value = value;  
    }  
}
```

```
class LongerNumber extends Number {  
    constructor(value: BigInt) {  
        super(value);  
    }  
}
```

Behavioral Subtyping

- Subtypes cannot have more restrictive (stronger) *pre-conditions*
 - That would prevent using the subclass as the parent-class
- But they can have *stronger post-conditions*
 - Not just in terms of return type

Is this behavioral subtyping?

```
class Rectangle {  
  
    int width;  
    int height;  
  
    public Rectangle(int width,  
                    int height) {  
        this.width = width;  
        this.height = height;  
    }  
}
```

```
public class Square extends Rectangle {  
  
    public Square(int width) {  
        super(width, width);  
    }  
}
```

Is this behavioral subtyping?

```
class Rectangle {  
  
    int width;  
    int height;  
  
    public Rectangle(int width,  
                    int height) {  
        this.width = width;  
        this.height = height;  
    }  
  
    // Sets just the width.  
    public void setWidth(int w) {  
        this.width = w;  
    }  
}
```

```
public class Square extends Rectangle {  
  
    public Square(int width) {  
        super(width, width);  
    }  
  
    public void setWidth(int w) {  
        this.width = w;  
        this.height = w;  
    }  
}
```

Behavioral Subtyping

- The compiler won't always check this for you
- There are many ways to enforce/restrict extension
 - abstract classes, can't be instantiated
 - But can have abstract methods that must be overridden
 - final methods, can't be overridden
 - Does not exist in TS
 - Heavily language-specific

JS/TS has Classes

Since ES2016

```
class Square {  
    width: number;  
  
    constructor(width: number) {  
        this.width = width;  
    }  
  
    printWidth() {  
        console.log(this.width);  
    }  
}  
  
let s1 = new Square(1);  
let s2 = new Square(2);  
s1.printWidth(); // 1  
s2.printWidth(); // 2
```

JS/TS has Classes

Since ES2016, but...

```
class Square {  
    width: number;  
  
    constructor(width: number) {  
        this.width = width;  
    }  
  
    printWidth() {  
        console.log(this.width);  
    }  
}
```

```
let s1 = new Square(1);  
let s2 = new Square(2);  
s1.printWidth(); // 1  
s2.printWidth(); // 2  
  
Square.prototype.printWidth = function () {  
    console.log('nope!');  
}  
  
s1.printWidth(); // 'nope'  
s2.printWidth(); // 'nope'
```

JS/TS has Classes

Since ES2016, but...

- No notion of static, private
 - TypeScript introduces keywords for these (and more).
- The definition of ‘this’ is tricky
 - Especially with inheritance
 - For those interested:
<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this>

Inheritance in JS/TS

```
class Animal {  
  
    private name: string;  
  
    constructor(name: string) {  
        this.name = name;  
    }  
}
```

```
class Dog extends Animal {  
  
    constructor() {  
        super("dog");  
    }  
  
    let dog = new Dog();  
    console.log(dog) // Dog { name: 'dog' }
```

So why inheritance?

- We already have interfaces; why not:

```
interface Rectangle {  
    getWidth(): number;  
    getHeight(): number;  
}  
  
class Square implements Rectangle {  
    width: number;  
    constructor(width: number) {  
        this.width = width;  
    }  
    getWidth(): number {  
        return this.width * this.width;  
    }  
    getHeight(): number { return getWidth(); }  
}
```

Inheritance vs. Subtyping

Inheritance is for polymorphism and code reuse

- Write code once and only once
- Superclass features implicitly available in subclass

```
class A extends B
```

Subtyping is for polymorphism

- Accessing objects the same way, but getting different behavior
- Subtype is substitutable for supertype

```
class A implements B  
class A extends B
```

So why inheritance?

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

```
class DebitCard implements PaymentCard {  
    private final String cardHolderName;  
    private final BigInteger digits;  
    private final Date expirationDate;  
    private int debit;  
  
    public DebitCard(String cardHolderName,  
                     BigInteger digits, Date expirationDate,  
                     int debit) {  
        this.cardHolderName = cardHolderName;  
        this.digits = digits;  
        this.expirationDate = expirationDate;  
        this.debit = debit;  
    }  
}
```

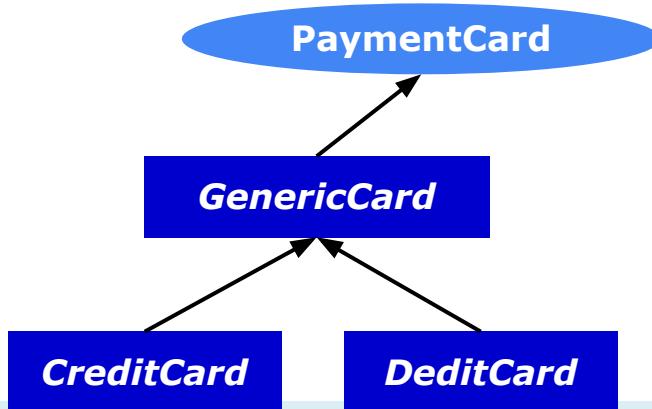
So why inheritance?

```
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;  
    }  
}
```

Inheritance Facilitates Reuse

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



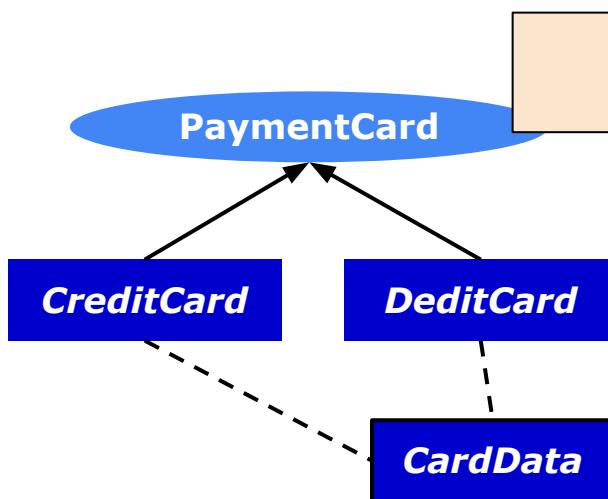
```
class GenericCard implements PaymentCard {  
    private final String cardHolderName;  
    private final BigInteger digits;  
    private final Date expirationDate;  
  
    public GenericCard(String cardHolderName,  
                       BigInteger digits, Date expirationDate) {  
        this.cardHolderName = cardHolderName;  
        this.digits = digits;  
        this.expirationDate = expirationDate;  
    }  
  
    @Override  
    public String getCardHolderName() {  
        return this.cardHolderName;  
    }  
}
```

Inheritance Facilitates Reuse

- When classes relate closely, it is nice to share functionality
 - That doesn't *necessitate* inheritance

Reuse does not Require Inheritance

```
public interface PaymentCard {  
    CardData getCardData();  
    int getValue();  
    boolean pay(int amount);  
}
```



Is this better?

```
class CardData {  
    private final String cardHolderName;  
    private final BigInteger digits;  
    private final Date expirationDate;  
  
    public CardData(String cardHolderName,  
                    Integer digits, Date expirationDate) {  
        this.cardHolderName = cardHolderName;  
        this.digits = digits;  
        this.expirationDate = expirationDate;  
    }  
  
    @Override  
    public String getCardHolderName() {  
        return this.cardHolderName;  
    }  
}
```

Reuse does not Require Inheritance

- When classes relate closely, it is nice to share functionality
 - That doesn't *necessitate* inheritance
- But inheritance can enable **substantial** reuse
 - When strong coupling is reasonable

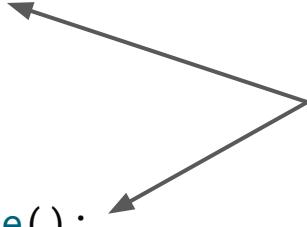
Template Method Pattern

```
class GiftCard implements PaymentCard {  
    private int balance;  
    public GiftCard(int balance) {  
        this.balance = balance;  
    }  
  
    @Override  
    public boolean pay(int amount) {  
        if (amount <= this.balance) {  
            this.balance -= amount;  
            return true;  
        }  
        return false;  
    }  
}
```

```
class DebitCard implements PaymentCard {  
    private int balance;  
    private int fee;  
    public DebitCard(int balance,  
                    int transactionFee) {  
        this.balance = balance;  
        this.fee = fee;  
    }  
  
    @Override  
    public boolean pay(int amount) {  
        if (amount <= this.balance) {  
            this.balance -= amount;  
            this.balance -= this.fee;  
            return true;  
        }  
        return false;  
    }  
}
```

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();
}
```



Must be implemented

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 Pattern

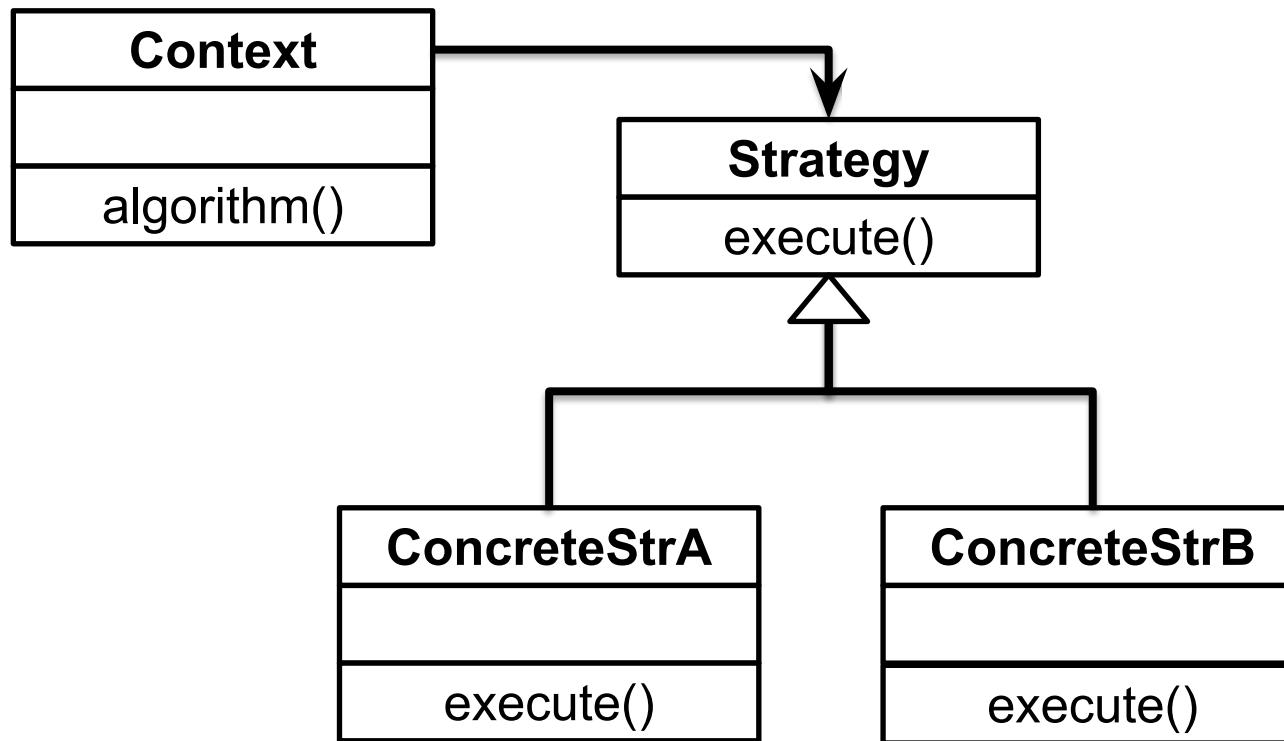
```
abstract class AbstractCashCard
    implements PaymentCard {
private int balance;
public AbstractCashCard(int balance) {
    this.balance = balance;
}
public boolean pay
    if (amount <= this.balance) {
        this.balance -= amount;
        chargeFee();
        return true;
    }
    return false;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

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

Design Tradeoffs?

```
class DebitCard extends AbstractCashCard
    @Override
    void chargeFee() {
        this.balance -= this.fee;
    }
}
```

Strategy Pattern



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

Inheritance vs. Composition + Delegation

- A lot of good design uses composition + delegation
 - Enables reuse, encapsulation by programming against interfaces
 - Composition facilitates adding multiple behaviors
 - Multiple inheritance exists, but gets messy
- Inheritance implies strong coupling
 - Sometimes a natural fit for reuse -- look for “is-a” relationships.
 - Much reduced encapsulation
 - Does not mean “no delegation”

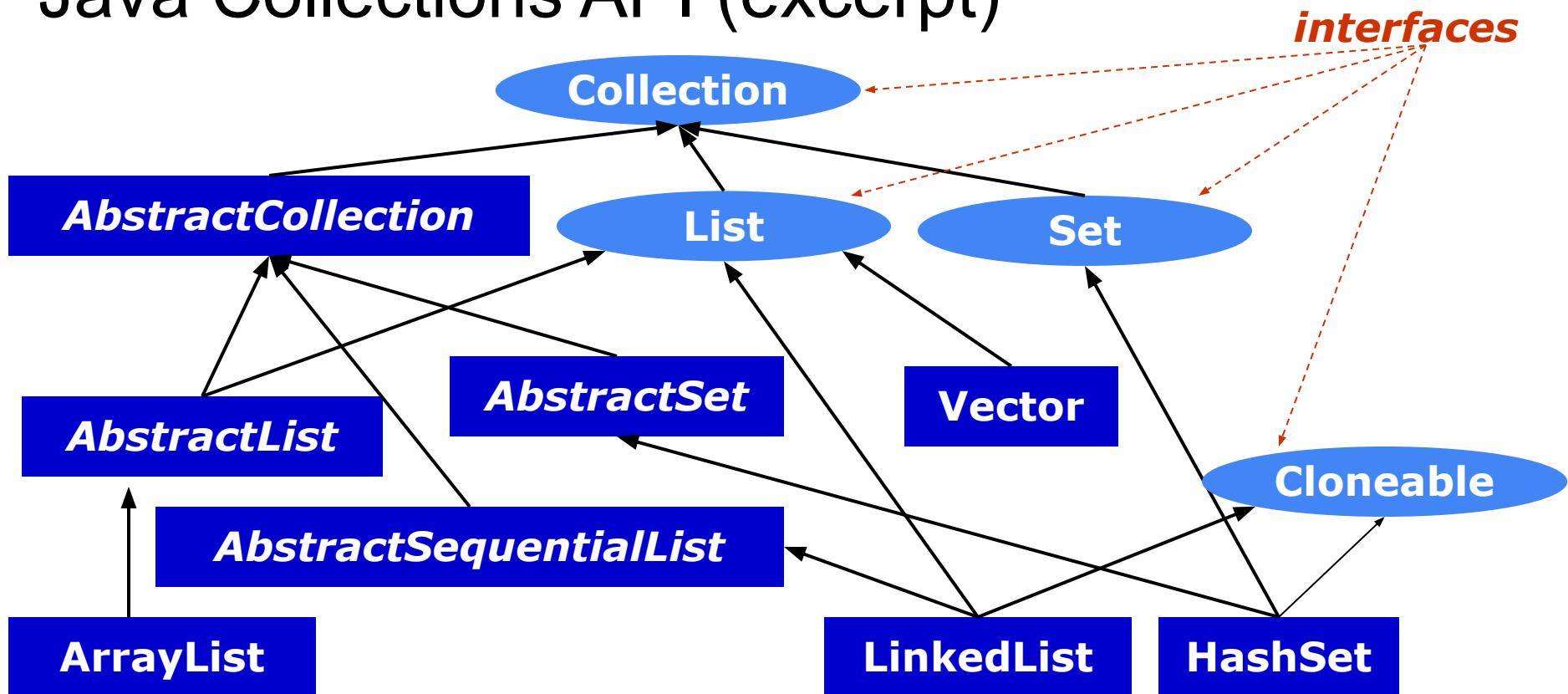
Inheritance vs. Composition + Delegation

- It's not an either/or question
 - Interfaces provide contracts
 - Inheritance provides reuse, strong coupling

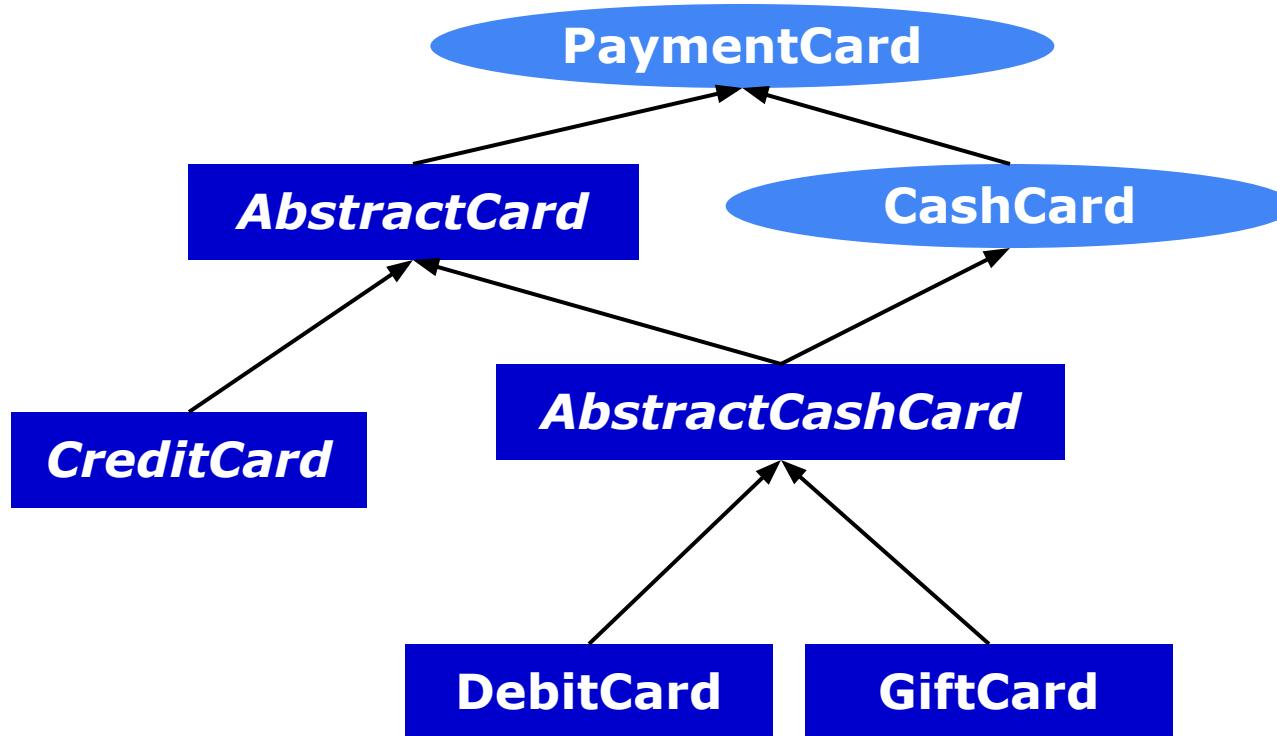
Interface Inheritance

```
public interface PaymentCard {  
    String getCardHolderName();  
    BigInteger getDigits();  
    Date getExpiration();  
    int getValue();  
    boolean pay(int amount);  
}  
  
interface CashCard extends PaymentCard {  
    boolean pay(int amount);  
    int getBalance();  
    void addCash(int amount);  
}
```

Java Collections API (excerpt)



Payment Card Hierarchy (example)



Payment Card with Inheritance

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

```
abstract class AbstractCard implements PaymentCard {  
    private final String cardHolderName;  
    private final BigInteger digits;  
    private final Date expirationDate;  
  
    public AbstractCard(String cardHolderName,  
                        BigInteger digits, Date expirationDate) {  
        this.cardHolderName = cardHolderName;  
        this.digits = digits;  
        this.expirationDate = expirationDate;  
    }  
  
    @Override  
    public String getCardHolderName() {  
        return this.cardHolderName;  
    }
```

Dynamic Dispatch

In Java:

- (Compile time) Determine which class to look in
- (Compile time) Determine method signature to be executed
 - Find all accessible, applicable methods
 - Select most specific matching method
- (Run time) Determine dynamic class of the receiver
- (Run time) From dynamic class, determine method to invoke
 - Execute method with the same signature found in step 2 (from dynamic class or one of its supertypes)

Language/Implementation Details

Details: final

- A final field: prevents reassignment to the field after initialization
- A final method: prevents overriding the method
- A final class: prevents extending the class
 - e.g., `public final class CheckingAccountImpl { ... }`
- Not present in TypeScript
 - Called “sealed” in some languages

Details: abstract

- An abstract method: must be overridden by a non-abstract subclass
- An abstract class: only classes allowed to have abstract members

Details: super

- Similar to `this`
- Refers to any (recursive) parent
 - Depending on what is accessed
- In TS, must call `super()`; before using ‘`this`’
 - Initializes the class
- In Java, `super` call needs to be first statement in constructor

Inheritance Reuse w/o Inversion of Control

```
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;
        return true;
    }
    return false;
}
```

```
class DebitCard extends AbstractCashCard
    @Override
    public boolean pay(int amount) {
        boolean success = super.pay(amount);
        if (success)
            this.balance -= this.fee;
        return success;
    }
}
```

Works because of the order of invocation.
But is it good?

Details: type-casting

- Sometimes you want a different type than you have

- e.g., double pi = 3.14;
int indianaPi = (int) pi;

In TS:

```
(dog as Animal).identify()
```

- 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

Designing with Inheritance in Mind

- Try to avoid it when composition+delegation is available
 - Delegation reduces coupling
 - Inheritance limits *information hiding*
- Document contracts for inheritance
 - The compiler won't enforce all invariants
- Enforce or prohibit inheritance where possible
 - In Java: `final` & `abstract`

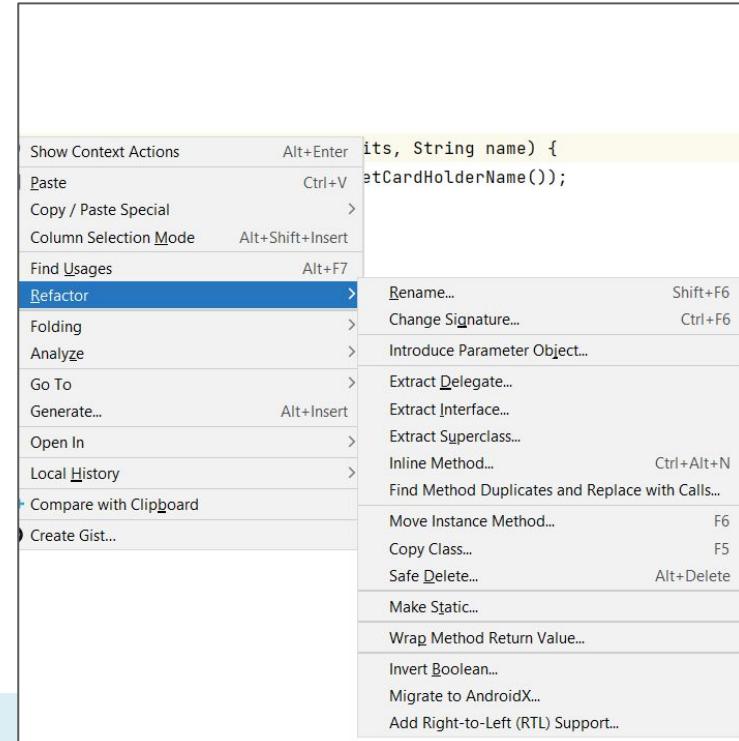
Refactoring

Refactoring

- Any functionality-preserving restructuring
 - Typically automated by IDE
 - Ideas?

Refactoring

- Rename class, method, variable to something not in-scope
- Extract method/inline method
- Extract interface
- Move method (up, down, laterally)
- Replace duplicates



Refactoring and Anti-Patterns

- Often, all the functionality is correct, but the organization is bad
 - High coupling, high redundancy, poor cohesion, god classes, ...
- Refactoring is the principal tool to improve structure
 - Automated refactorings even guarantee correctness
 - But you can't always count on those being right
 - A series of refactorings is usually enough to introduce design patterns

Refactoring and Anti-Patterns

- Often, all the functionality is correct, but the organization is bad
 - High coupling, high redundancy, poor cohesion, god classes, ...
- Refactoring is the principal tool to improve structure
 - Automated refactorings even guarantee correctness
 - But you can't always count on those being right
 - A series of refactorings is usually enough to introduce design patterns
- HW4 involves analyzing such a system and making primarily refactoring changes
 - “primarily”, because sometimes you do need to alter things slightly.

Summary

- Inheritance is a powerful tool
 - That takes coupling to the extreme
 - And deserves careful consideration
 - Template method pattern enforces reuse, limits customization
- Subtyping and inheritance are related, but not the same
 - Composition & Delegation are often the right tools
 - Not mutually exclusive

Principles of Software Construction: Objects, Design, and Concurrency

Refactoring & Anti-patterns

Christian Kästner

Vincent Hellendoorn



HW3 Feedback

<https://rb.gy/xpnh1b>



Today

- Midterm debrief
 - Discussing the Decorator Pattern
- Revisiting composition + delegation
- More on inheritance
- Refactoring and Anti-patterns

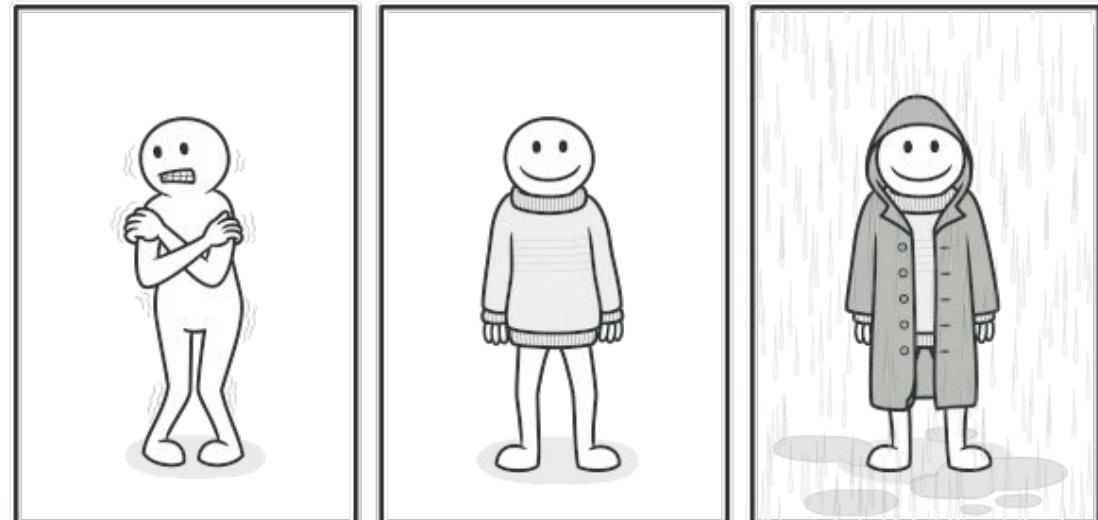
Midterm Debrief

How did it go? Anything deserve discussion?

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:



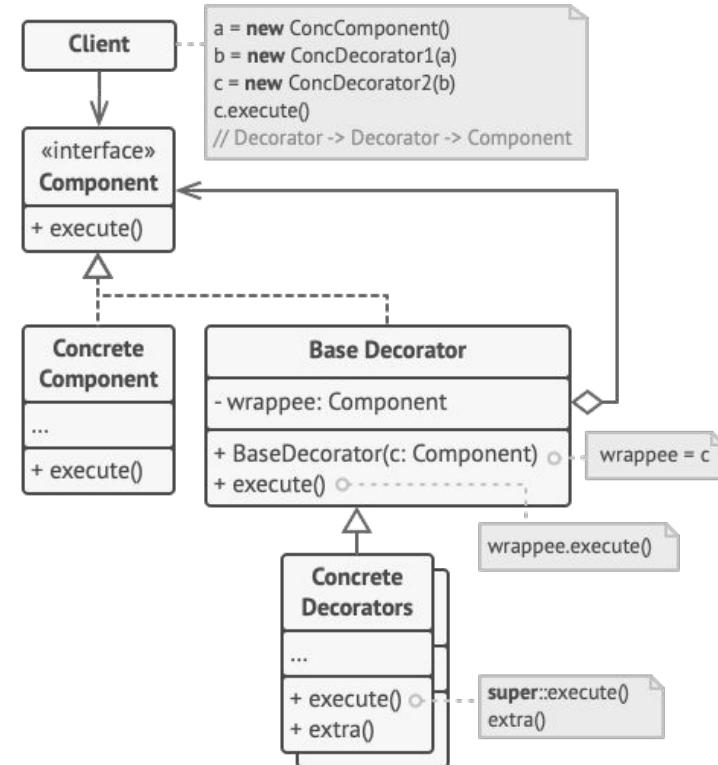
Drawing Example -- Basics

```
class Line {  
    // TODO  
}  
  
interface Shape {  
    toLines(): Line[];  
}  
  
class Triangle implements Shape {  
    public toLines(): Line[] {  
        return ...;  
    }  
}
```

Drawing Example -- Basics

```
// A drawing consists of many shapes.  
class Drawing {  
    shapes: Shape[]  
    constructor(shapes: Shape[]) {  
        this.shapes = shapes;  
    }  
    public toLines() {  
        let lines: Line[] = []  
        for (let shape of this.shapes) {  
            lines.push(shape.toLines());  
        }  
        return lines;  
    }  
}
```

Drawing Example -- Decorator?



Drawing Example -- Decorator?

```
interface DrawingSaver {  
    saveDrawing(drawing: Drawing, path: string): void;  
}  
  
class BasicSaver implements DrawingSaver {  
    public saveDrawing(drawing: Drawing, path: string): void {  
        let lines: Line[] = drawing.toLines();  
        // Now what?  
    }  
}
```

Drawing Example -- Decorator?

```
class DrawingSaverDecorator implements DrawingSaver {  
    wrappee: DrawingSaver  
    constructor(source: DrawingSaver) { this.wrappee = source; }  
  
    public saveDrawing(drawing: Drawing, path: string): void {  
        this.wrappee.saveDrawing(drawing, path);  
    }  
}  
  
class JPEGDecorator extends DrawingSaverDecorator {  
    public saveDrawing(drawing: Drawing, path: string): void {  
        let lines: Line[] = drawing.toLines();  
        // Internally store in JPEG  
        super.saveDrawing(drawing, path);  
    }  
}
```

Drawing Example -- Strategy

```
interface LineFormatter {  
    write(lines: Line[], writer: Writer): void;  
}  
  
class DrawingSaver {  
    public save(drawing: Drawing, formatter: LineFormatter, path: string) {  
        let lines: Line[] = drawing.toLines();  
        let writer: Writer = new Writer(path);  
        formatter.write(lines, writer);  
    }  
}  
  
class JPEGFormat implements LineFormatter {  
    public write(lines: Line[], writer: Writer) { // Store JPEG data. }  
}
```

Drawing Example -- Template Method

```
abstract class DrawingSaver {  
    public save(drawing: Drawing, path: string) {  
        let lines = drawing.toLines();  
        let formatted = this.toFormat(lines);  
        let writer: Writer = new Writer(path);  
        writer.write(formatted);  
    }  
  
    abstract toFormat(lines: Line[]): any[];  
}  
  
class JPEGSaver extends DrawingSaver {  
    public toFormat(lines: Line[]): any[] { // Store JPEG data. }  
}
```

Today

- Midterm debrief
 - Discussing the Decorator Pattern
- **Revisiting composition + delegation**
- More on inheritance
- Refactoring and Anti-patterns

Delegation

- *Delegation* is simply when one object relies on another object for some subset of its functionality
 - e.g. here, the Sorter is delegating functionality to some Order
- Judicious delegation enables code reuse

```
interface Order {  
    boolean lessThan(int i, int j);  
}  
final Order ASCENDING = (i, j) -> i < j;  
final Order DESCENDING = (i, j) -> i > j;  
  
static void sort(int[] list, Order cmp) {  
    ...  
    boolean mustSwap =  
        cmp.lessThan(list[i], list[j]);  
    ...
```

Using delegation to extend functionality

- Consider the `java.util.List` (excerpted):

```
public interface List<E> {  
    public boolean add(E e);  
    public E      remove(int index);  
    public void   clear();  
  
    ...  
}
```

- Suppose we want a list that logs its operations to the console...

Using delegation to extend functionality

```
public class LoggingList<E> implements List<E> {  
    private final List<E> list;  
    public LoggingList<E>(List<E> list) { this.list = list; }  
  
    public boolean add(E e) {  
        System.out.println("Adding " + e);  
        return list.add(e);  
    }  
  
    public E remove(int index) {  
        System.out.println("Removing at " + index);  
        return list.remove(index);  
    }  
}
```

The `LoggingList` is composed of a `List`, and delegates (the non-logging) functionality to that `List`

Using inheritance to extend functionality

```
public class LoggingList<E> extends ??? {  
  
    public boolean add(E e) {  
        System.out.println("Adding " + e);  
        return super.add(e);  
    }  
  
    public E remove(int index) {  
        System.out.println("Removing at " + index);  
        return super.remove(index);  
    }  
}
```

Using inheritance to extend functionality

```
public class ArrayLoggingList<E> extends ArrayList<E> {  
  
    public boolean add(E e) {  
        System.out.println("Adding " + e);  
        return super.add(e);  
    }  
  
    public E remove(int index) {  
        System.out.println("Removing at " + index);  
        return super.remove(index);  
    }  
}
```

The LoggingList is an ArrayList,
and relies on it for the (the
non-logging) functionality.

Delegation and Design

- Small interfaces with clear contracts
- Classes to encapsulate algorithms, behaviors
 - E.g., the Order

Designing with Inheritance in Mind

- Try to avoid it when composition+delegation is available
 - Delegation reduces coupling
 - Inheritance limits *information hiding*
- Document contracts for inheritance
 - The compiler won't enforce all invariants
- Enforce or prohibit inheritance where possible
 - In Java: `final` & `abstract`

Today

- Midterm debrief
 - Discussing the Decorator Pattern
- Revisiting composition + delegation
- **More on inheritance**
- Refactoring and Anti-patterns

Details: this

- Refers to itself, but what is that?
 - In general, behaves as a variable referencing the current object.
 - Knows names of all fields and methods
 - Including private ones, and ones inherited from parents
 - But not generally types, modifiers
 - This is where *reflection* comes in, which is significantly easier in some languages
 - E.g., in Python: `self.__dict__` is all field names on self
 - In JS, `object.__proto__` does something similar
- The definition of ‘this’ can get murky with inheritance
 - Subtle differences between Java and JS/TS, Python

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();

What is ‘this’?

```
class BaseSaver implements DrawingSaver {  
    public save(Drawing drawing, Writer writer) {  
        Line[] lines = drawing.toLines();  
        this.serialize(writer, lines);  
    }  
    public serialize(writer, lines) { ... }  
}  
  
class PNGDrawingSaver implements DrawingSaver  
DrawingSaver delegate;  
  
public save(Drawing drawing, Writer writer) {  
    delegate.save(drawing, writer);  
}  
public serialize(...) { // write PNG }
```

Which ‘serialize’ gets called?

```
new PNGDrawingSaver(  
    new BaseSaver()).save(...);
```

Details: super

- Similar to this
- Refers to any (recursive) parent
 - Depending on what is accessed
- In TS, must call super(); before using ‘this’
 - Initializes the class
- In Java, super call needs to be first statement in constructor

Details: final

- A final field: prevents reassignment to the field after initialization
- A final method: prevents overriding the method
- A final class: prevents extending the class
 - e.g., `public final class CheckingAccountImpl { ... }`
- Not present in TypeScript
 - Called “sealed” in some languages

Details: abstract

- Method: must be overridden by a non-abstract subclass
- Class: only classes allowed to have abstract members

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()`

Today

- Midterm debrief
 - Discussing the Decorator Pattern
- Revisiting composition + delegation
- More on inheritance
- **Refactoring and Anti-patterns**

Recall: Refactoring and Anti-Patterns

- Often, all the functionality is correct, but the organization is bad
 - High coupling, high redundancy, poor cohesion, god classes, ...
- Refactoring is the principal tool to improve structure
 - Automated refactorings even guarantee correctness
 - But you can't always count on those being right
 - A series of refactorings is usually enough to introduce design patterns
- HW4 involves analyzing such a system and making primarily refactoring changes
 - “primarily”, because sometimes you do need to alter things slightly.

Anti-patterns

Anti-patterns are *common* forms of bad/no-design

- Can you think of examples?
- Where do they come from?

Anti-patterns

- We have talked a fair bit about bad design heuristics
 - High coupling, low cohesion, law of demeter, ...
- You will see a much larger vocabulary of related issues
 - Commonly called code/design “smells”
 - Worth reads:
 - A short overview: <https://refactoring.guru/refactoring/smells>
 - Wikipedia: https://en.wikipedia.org/wiki/Anti-pattern#Software_engineering
 - Book on the topic (no required reading): Refactoring for Software Design Smells: Managing Technical Debt, Suryanarayana, Samarthym and Sharma
 - S.O. summary: <https://stackoverflow.com/a/27567960>

Anti-patterns

- Two ways of looking at this:
 - Design issues that manifest as bad/unmaintainable code
 - Poorly written/evolved code that leads to bad design
 - Next two slides show both

Anti-patterns

- Common system-level anti-patterns
 - Bad encapsulation, violates information hiding
 - public fields should be private; interface leaks implementation details; lack of interface
 - Bad modularization, violates coupling
 - related methods in different places, or vice versa; very large interface; “god” class
 - Bad abstraction, violates cohesion
 - Not exposing relevant functionality; near-identical classes; too many responsibilities
 - Bad inheritance/hierarchy
 - Violating behavioral subtyping; unnecessary inheritance; very large hierarchies (too wide or too deep)

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)

Anti-patterns

- Zooming in: common code smells
 - Not necessarily bad, but worthwhile indicators to check
 - When problematic, often point to design problems
 - Excessive, unused hierarchies
 - Operations posing as classes
 - Data classes
 - Tricky: not always bad, but ideally distinguish from regular classes (e.g., ‘record’), and assign responsibilities if any exist (think: FlashCard did equality checking)
 - Heavy usage of one class’ data from another (“feature envy”, “inappropriate intimacy”; poor coupling)
 - Long chains of calls needed to do anything (law of demeter)
 - A class that only delegates work

Anti-patterns

- You can detect them from either side
 - Pick a design principle, look for violations
 - Identify “weird” code and isolate design flaw

Anti-patterns

- You can detect them from either side
 - Pick a design principle, look for violations
 - Identify “weird” code and isolate design flaw
- All fairly easy to spot on their own
 - But in HW4, there are multiple, tangled up
 - We actually provide way more guidance than you'll get in the wild!
 - How do you approach that?

Refactoring and Anti-patterns

- Identifying multiple design problems
 - Make a list
 - Read the code, record anything that stands out
 - Pay attention to class names and their (apparent) interfaces
 - Make note of repetitive code (esp. across methods)
 - Draw a diagram, using a tool or by hand
 - Spot duplication, (lack of) interfaces, strange inheritance
 - This takes **practice**
 - Don't solve every problem
 - Many issues are orthogonal
 - Or, at least, you can improve things somewhat
 - When issues intersect, prioritize fixing interfaces

Refactoring

- So where is “refactoring” in all this?
 - It’s what comes next.
 - Most design issues can be resolved with functionality-preserving transformation(s)
 - Too many parameters? Merge relevant ones into object, and/or replace with method calls.
 - Two near-identical classes? Merge their signatures using renamings, parameterization, then delete one or extract super-class
 -

Summary

- Practice applying design patterns, recognizing anti-patterns
 - Create scenarios and try to write code
 - Find examples in public projects
 - We'll do a case-study on Thursday
- Use this time to gain experience
 - Read lots of code, think about alternatives, like in HW4
 - Learn a vocabulary of anti-patterns (even if imperfect)

Principles of Software Construction: Objects, Design, and Concurrency

Asynchrony and Concurrency

Christian Kästner

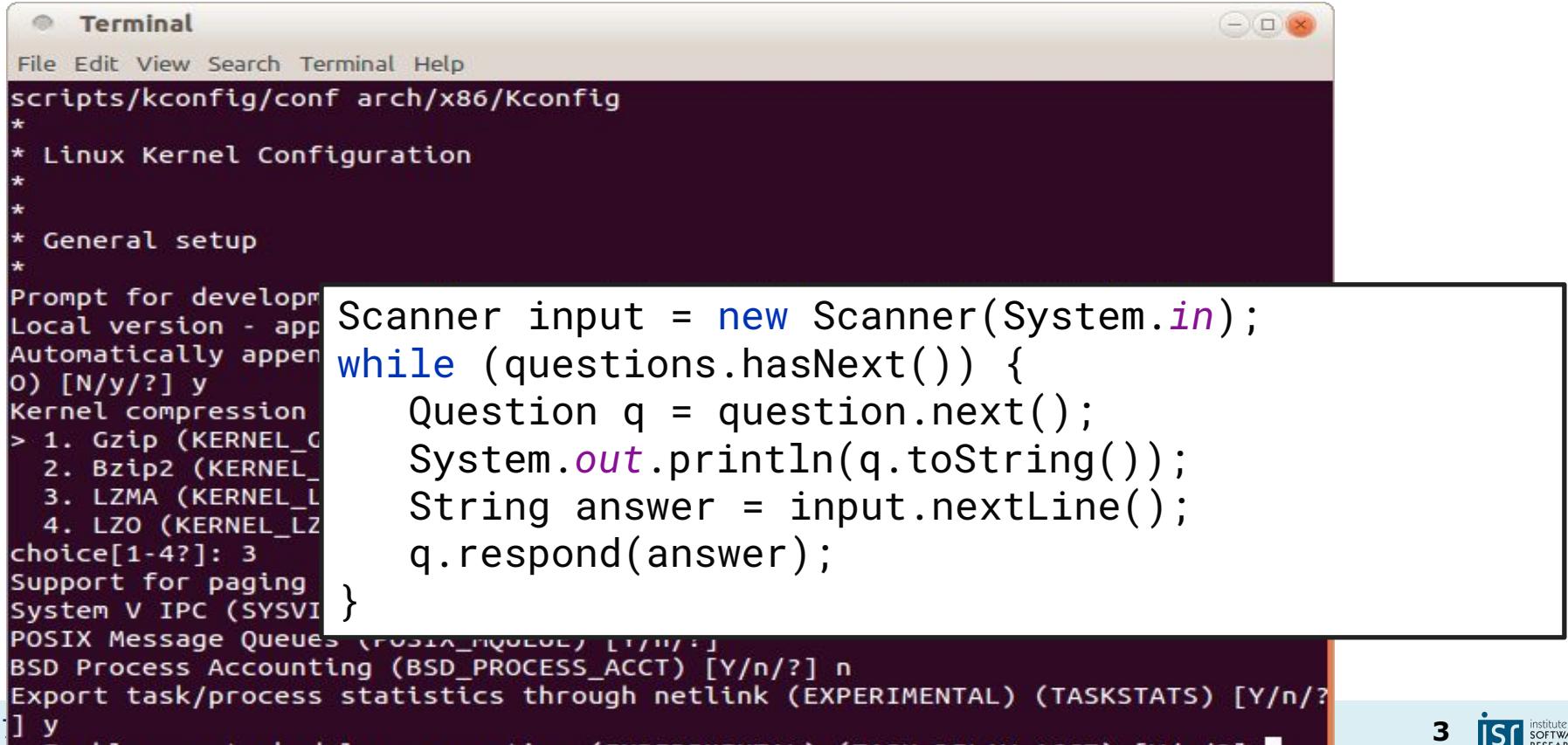
Vincent Hellendoorn



How was the Recitation?

- Did every solution make the program smaller?
- Did I change everything you would have?
 - Anything you wouldn't?

Interaction with CLI

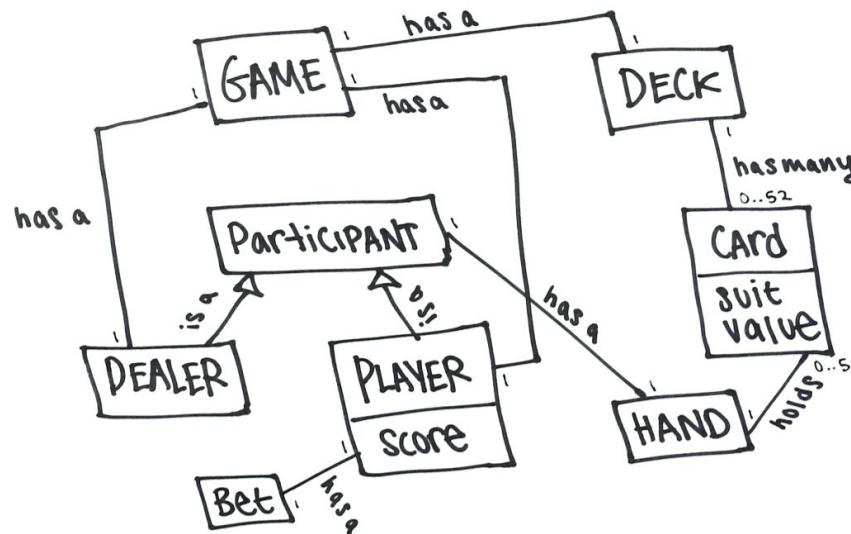


Terminal

```
File Edit View Search Terminal Help
scripts/kconfig/conf arch/x86/Kconfig
*
* Linux Kernel Configuration
*
*
* General setup
*
Prompt for developer options [Y/n/?] n
Local version - appname [v1.0] v1.0
Automatically append version number [Y/n/?] n
0) [N/y/?] y
Kernel compression type
  1. Gzip (KERNEL_GZIP)
  2. Bzip2 (KERNEL_BZIP2)
  3. LZMA (KERNEL_LZMA)
  4. LZO (KERNEL_LZO)
choice[1-4?]: 3
Support for paging model [SWIPE]
System V IPC (SYSVIPC)
POSIX Message Queues (POSIX_MESSAGEQUEUES) [Y/n/?] n
BSD Process Accounting (BSD_PROCESS_ACCT) [Y/n/?] n
Export task/process statistics through netlink (EXPERIMENTAL) (TASKSTATS) [Y/n/?] n
1) y
Enable per-task delay accounting (EXPERIMENTAL) (TASK_DELAY_ACCT) [Y/n/?] n
```

```
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);
}
```

A backend with no interaction



One Possible
Domain model

this is NOT a reference solution, it's
an example of what a domain model
looks like

What have we not yet seen?

How do you wait?

The screenshot shows a Microsoft Visual Basic for MS-DOS interface. The menu bar includes File, Edit, View, Search, Run, Debug, Options, Window, and Help. The Help menu is open, displaying options: Index, Contents, Keyboard, Topic: Using Help F1, Tutorial, and About... A context menu is overlaid on the code editor window, containing the same items as the Help menu. The code editor window title is [1] CLOCK.FRM. The code in the editor is:

```
' Include file containing declaration  
'$INCLUDE: 'clock.bi'  
  
CONST FALSE = 0  
CONST TRUE = NOT FALSE  
CONST ALARMSOUND = "MBT255L1606C04GED"  
  
DIM SHARED AlarmTime AS STRING  
DIM SHARED TimeFmt AS STRING  
  
{  
    ...  
    }  
F1=Help | Display version number, cop
```

The context menu is highlighted with a black border and contains the following code:

```
while (true) {  
    if (isKeyDown("Alt+Q"))  
        break;  
    if (isKeyDown("F1"))  
        openHelp();  
    if (isMouseDown(10 ...))  
        startMovingWindow();  
    ...  
}
```

How do you multi-player?



```
while (true) {  
    if (player === "player1") {  
        hasWon = play("player1");  
        if (hasWon) break;  
        player = "player2";  
    } else (player === "player2") {  
        hasWon = play("player2")  
        if (hasWon) break;  
        player = "player1";  
    }  
}
```

Today

Beyond serial execution

- Event-based Programming
- Asynchrony & Concurrency
- I/O, GUIs
- Observer Pattern
- React preview

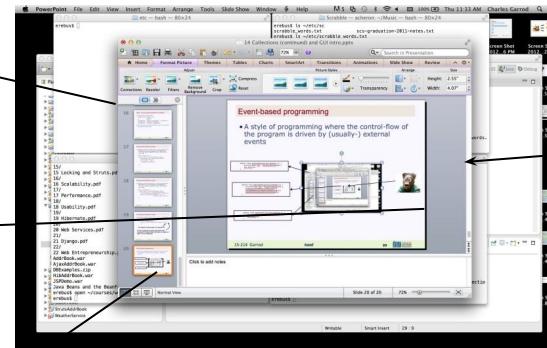
Event-based programming

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

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

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

```
public void performAction(ActionEvent e) {  
    List<String> lst = Arrays.asList(bar);  
    foo.peek(40)  
}
```



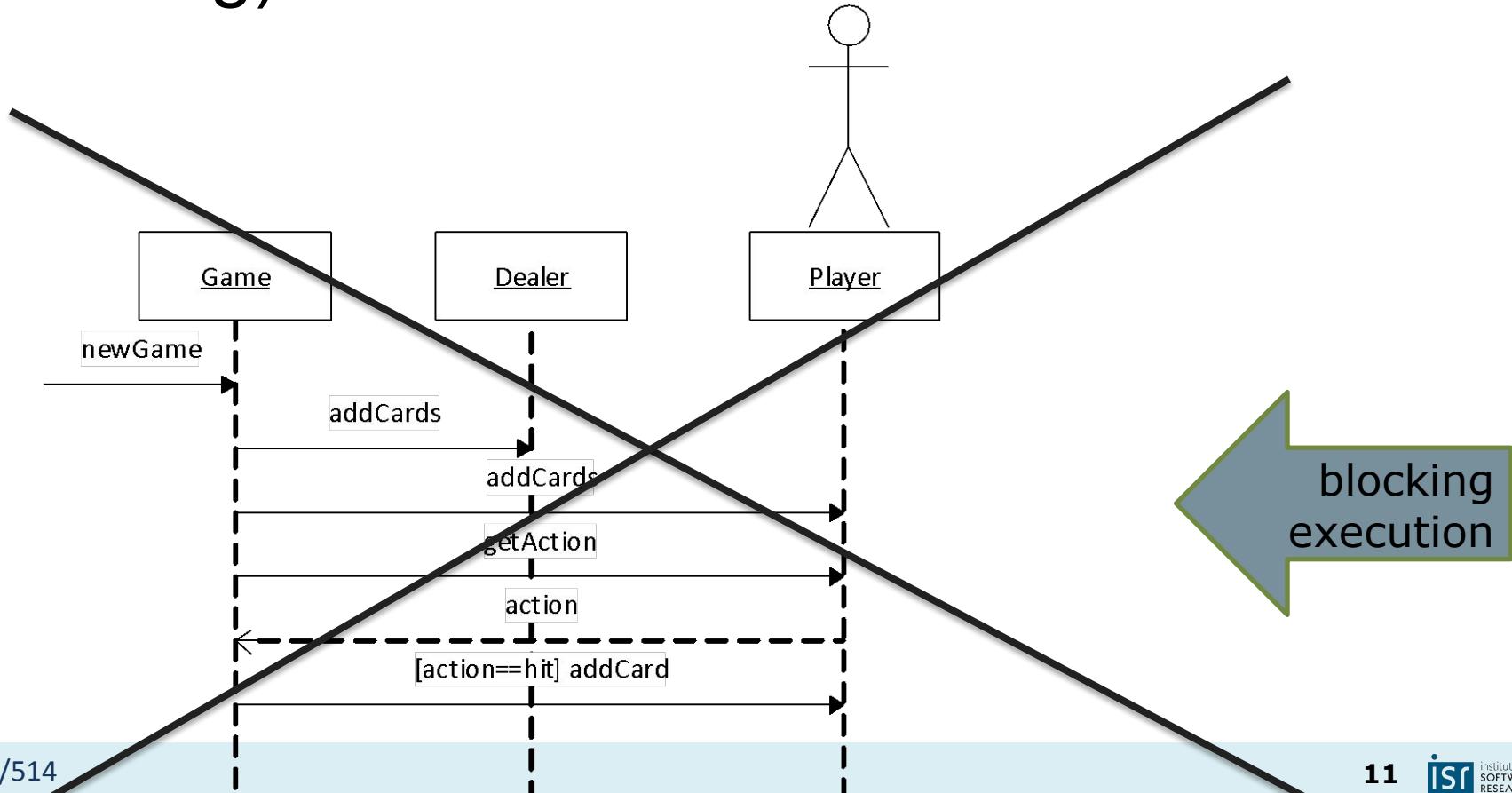
Event-based GUIs

//static public void main...
JFrame window = ...
window.setDefaultCloseOperation(
 WindowConstants.EXIT_ON_CLOSE);
window.setVisible(true);

//on add-button click:
String email = emailField.getText();
emaillist.add(email);

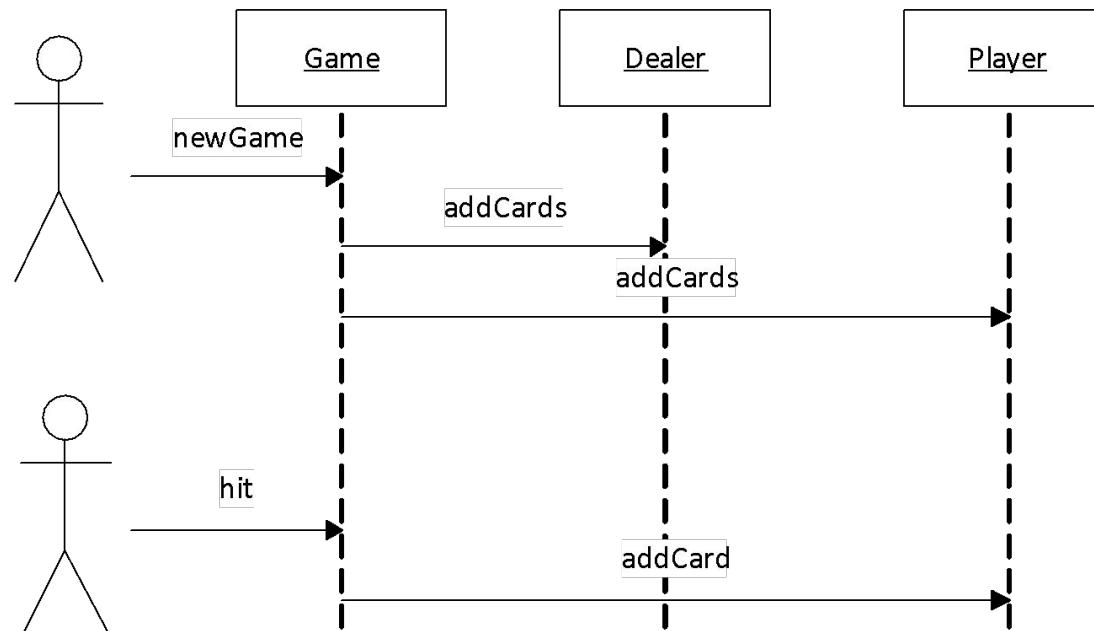
//on remove-button click:
int pos = emaillist.getSelectedItem()
if (pos>=0) emaillist.delete(pos);

(Blocking) Interactions with users



Interactions with users through events

- Do not block waiting for user response
- Instead, react to user events



Three Concepts of Importance

- Thread: instructions executed in sequence
 - Within a thread, everything happens in order.
 - A thread can start, sleep, and die.
 - You often work on the “main” thread.

Three Concepts of Importance

- Thread: instructions executed in sequence
 - Within a thread, everything happens in order.
 - A thread can start, sleep, and die.
 - You often work on the “main” thread.
- Concurrency: multiple threads running at the same time
 - Not necessarily *executing* in parallel

Three Concepts of Importance

- Thread: instructions executed in sequence
 - Within a thread, everything happens in order.
 - A thread can start, sleep, and die.
 - You often work on the “main” thread.
- Concurrency: multiple threads running at the same time
 - Not necessarily *executing* in parallel
- Asynchrony: computation happening outside the main flow

Multi-Threading

The natural response to non-serial computation

- Multiple threads can exist concurrently
- Threads share memory space
- You are already using it
 - Garbage collection in the JVM

Asynchrony

Where might this come from?

Asynchrony

Where might this come from?

- People
- Other machines
- Our own *callbacks*

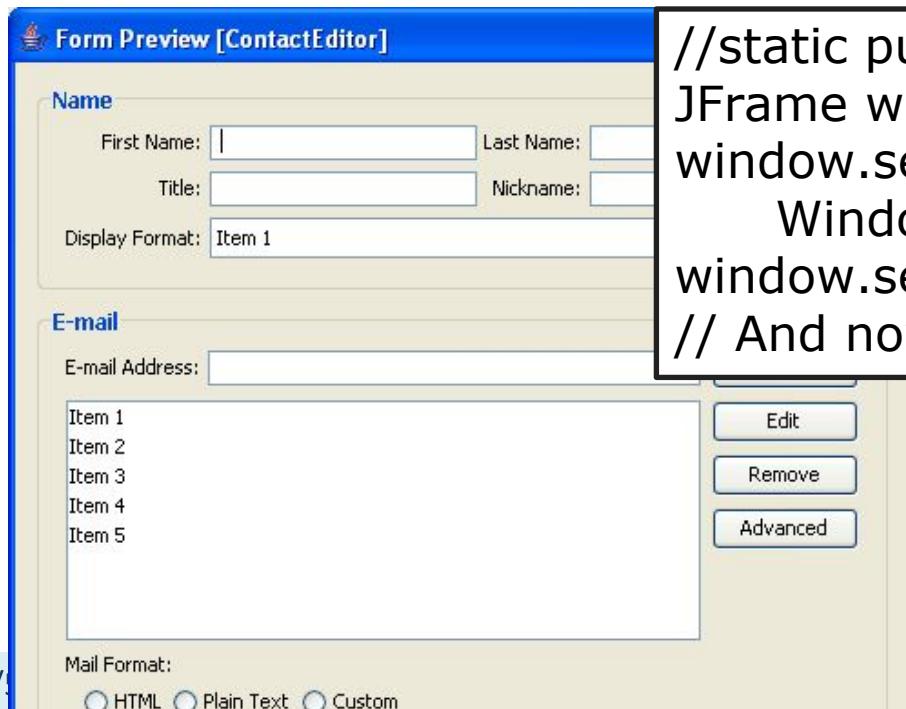
Asynchrony

Usually, managing asynchronous events involves concurrency

- Do something while we wait
- Multiple events can overlap
- Even “waiting” is not really doing nothing
- We will focus on constructs for handling both

Asynchrony

Asynchronous but not concurrent



```
//static public void main...
JFrame window = ...
window.setDefaultCloseOperation(
    WindowConstants.EXIT_ON_CLOSE);
window.setVisible(true);
// And now, wait.
```

Where do we want concurrency?

Where do we want concurrency?

- User interfaces
 - Events can arrive any time
- File I/O
 - Offload work to disk/network/... handler

Where do we want concurrency?

- Background work
 - Periodically run garbage collection, check health of service
- High-performance computing
 - Facilitate parallelism and distributed computing

User Interfaces

What happens here:

```
document.addEventListener('click', () => console.log('Clicked!'))
```

User Interfaces

Callback functions

- Perhaps *the* building blocks of the internet's UI.
- Work that should be done once something happens
 - Called asynchronously from the literal flow of the code
 - Not concurrent: JS is single-threaded

```
document.addEventListener('click', () => {  
  console.log('Clicked!'); console.log('Clicked again!'); })
```

Concurrency with file I/O

Key chart:

Computer Action	Avg Latency	Normalized Human Time
3GhzCPU Clock cycle 3Ghz	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
RAM access	70 - 100ns	3.5 to 5.5 min
NVMe SSD I/O	7-150 µs	2 hrs to 2 days
Rotational disk I/O	1-10 ms	11 days to 4 mos
Internet: SF to NYC	40 ms	1.2 years
Internet: SF to Australia	183 ms	6 years
OS virtualization reboot	4 s	127 years
Virtualization reboot	40 s	1200 years
Physical system reboot	90 s	3 Millenia

Table 1: Computer Time in Human Termsⁱ

<https://formulusblack.com/blog/compute-performance-distance-of-data-as-a-measure-of-latency/>

Concurrency with file I/O

Mostly used synchronous IO so far

```
/**  
 * in the top-level directory only look for subdirectories and metadata files  
 */  
processProject (builder: ProjectBuilder, dir: string): void {  
    const files = fs.readdirSync(dir)  
    for (const filename of files) {  
        const file = path.join(dir, filename)  
        const fileStats = fs.statSync(file)  
        const extension = path.extname(file)  
        if (fileStats.isDirectory()) { this.#processDirectory(builder, file) }  
        else if (extension === '.yml') { this.#loadMetadataFile(builder, file) }  
    }  
}
```

Concurrency with file I/O

Mostly used synchronous IO so far

- Works fine if ‘fetch’ is synchronous
 - But if other work is waiting...

```
let image: Image = fetch('myImage.png');  
display(image);
```

Concurrency with file I/O

Mostly used synchronous IO so far

- Works fine if ‘fetch’ is synchronous
 - But if other work is waiting...

```
let image: Image = fetch('myImage.png');
display(image);
```
- It'd be nice if we could continue other work
 - How to make it work if ‘fetch’ is asynchronous?

Concurrency with file I/O

Asynchronous code requires Promises

- Captures an intermediate state
 - Neither fetched, nor failed; we'll find out eventually

```
let imageToBe: Promise<Image> = fetch('myImage.png');
imageToBe.then((image) => display(image))
    .catch((err) => console.log('aw: ' + err));
```

Concurrency with file I/O

Asynchronous code requires Promises

- Captures an intermediate state
 - Neither fetched, nor failed; we'll find out eventually

```
let imageToBe: Promise<Image> = fetch('myImage.png');
imageToBe.then((image) => display(image))
    .catch((err) => console.log('aw: ' + err));
```

- A *bit* like a callback
 - But better designed
 - Also related to async/await
 - Future in Java

Concurrency with file I/O

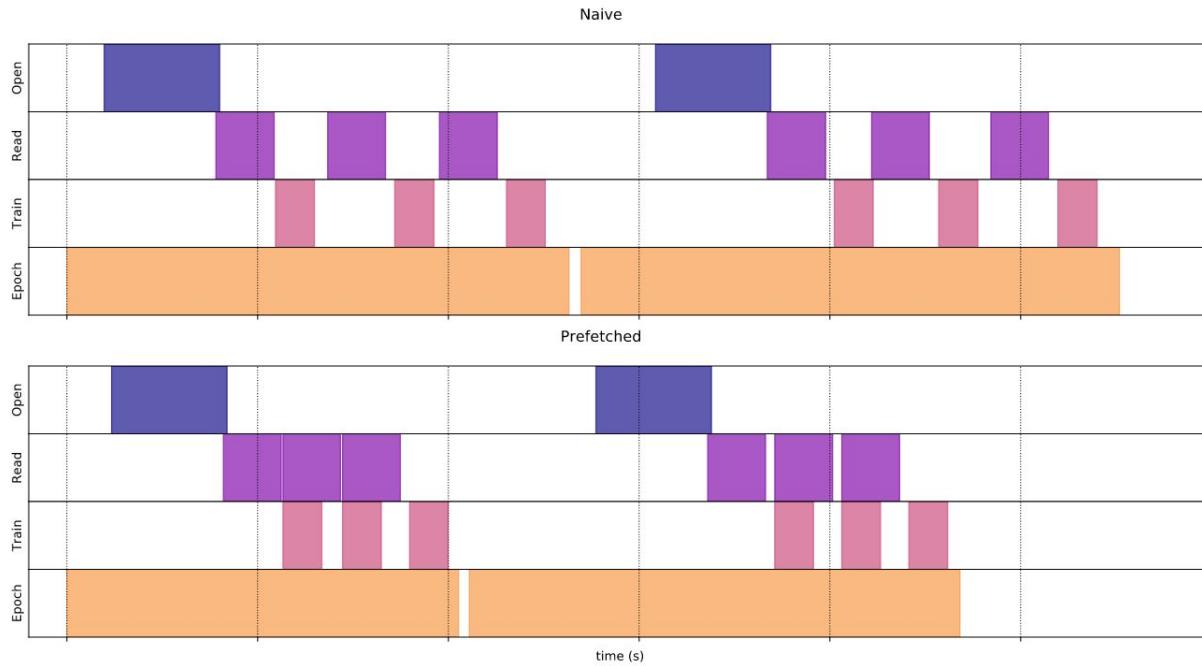
Can save you a lot of time

- An example from Machine Learning
- The usual process:
 - Read data from a filesystem or network
 - Batch samples, send to GPU/TPU/XPU memory
 - Train on-device

Concurrency with file I/O

An example from Machine Learning

Different devices:



Aside: Concurrency vs. parallelism

- Concurrency without parallelism:



- Concurrency with parallelism:



Aside: Threads vs. Processes

- Threads are lightweight; processes heavyweight
- Threads share address space; processes have own
- Threads require synchronization; processes don't
 - Threads hold locks while mutating objects
- It's unsafe to kill threads; safe to kill processes

Concurrency

Quite a few advanced topics

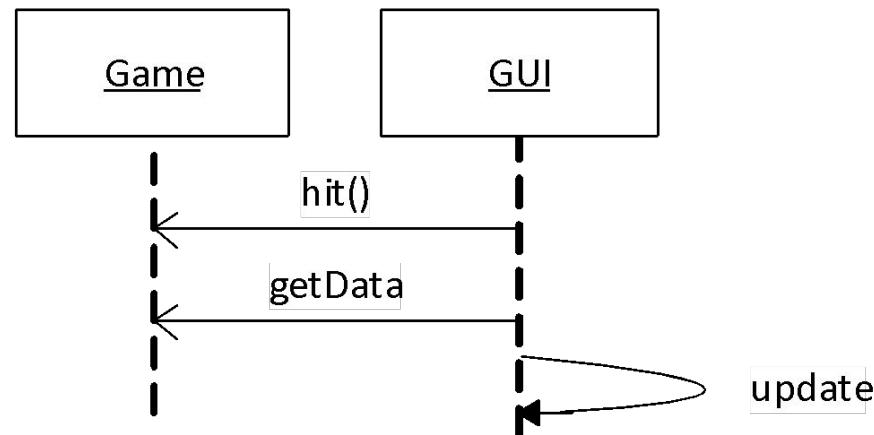
- Synchronization
- Immutability
- Parallelism
- More later in the course
 - Except for parallelism; largely out of scope

Designing for Asynchrony & Concurrency

- We are in a new paradigm now
 - We need standardized ways to handle asynchronous and/or concurrent interactions
 - This is how design patterns are born
- A lot of powerful syntax for managing concurrency
 - To be discussed in future classes

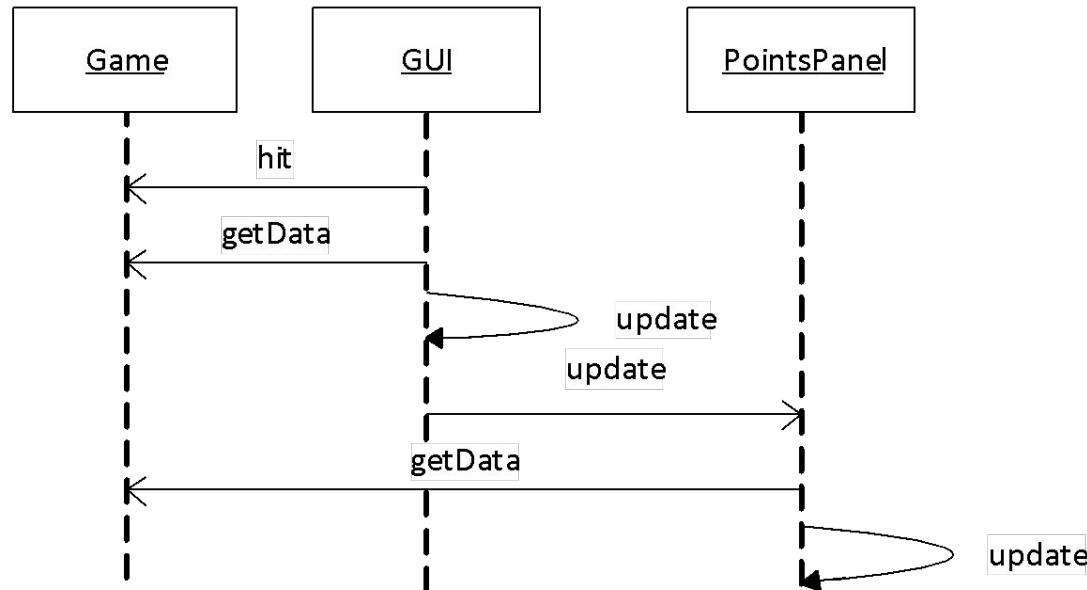
A GUI design challenge

- Consider a blackjack game, implemented by a Game class:
 - Player clicks “hit” and expects a new card
 - When should the GUI update the screen?



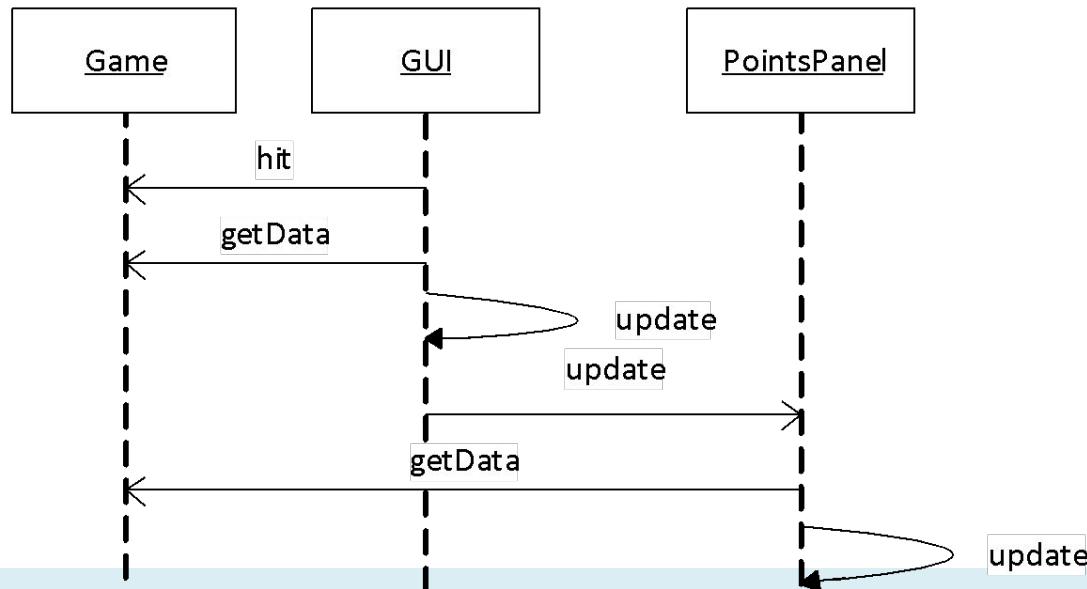
A GUI design challenge, extended

- What if we want to show the points won?



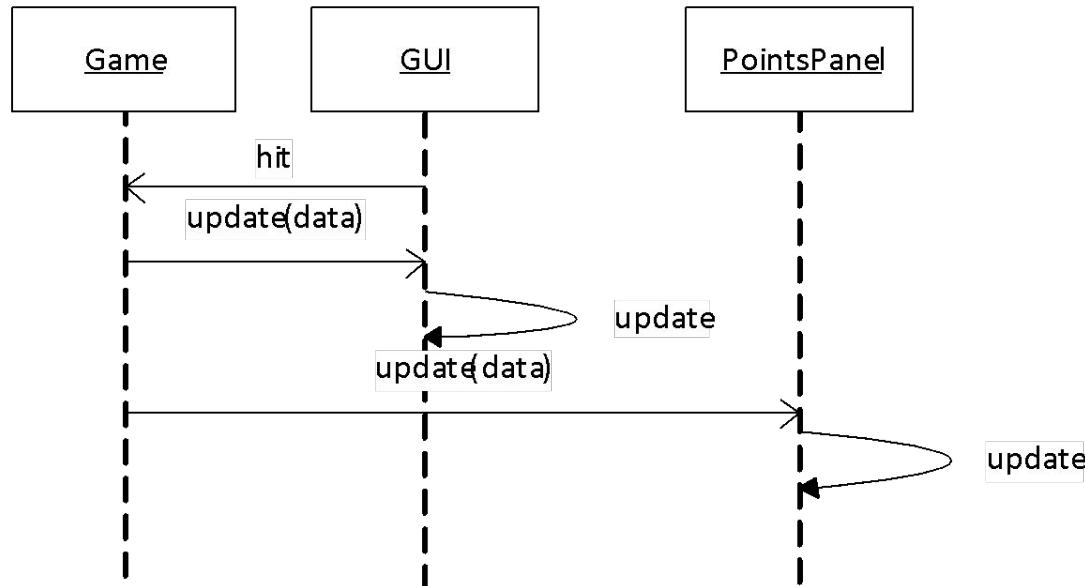
Game updates GUI?

- What if points change for reasons not started by the GUI?
(or computations take a long time and should not block)



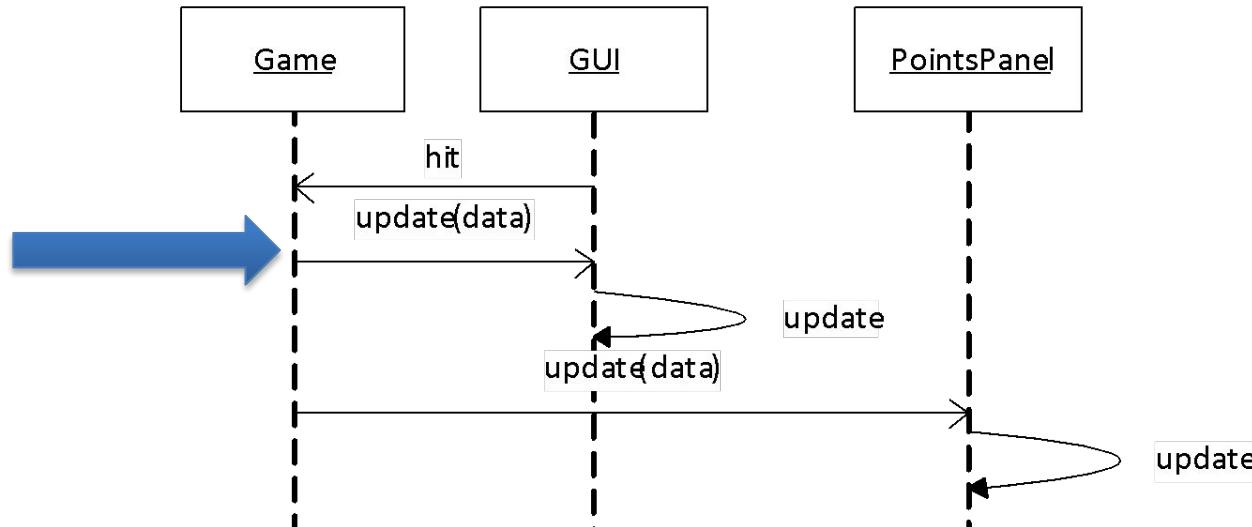
Game updates GUI?

- Let the Game tell the GUI that something happened



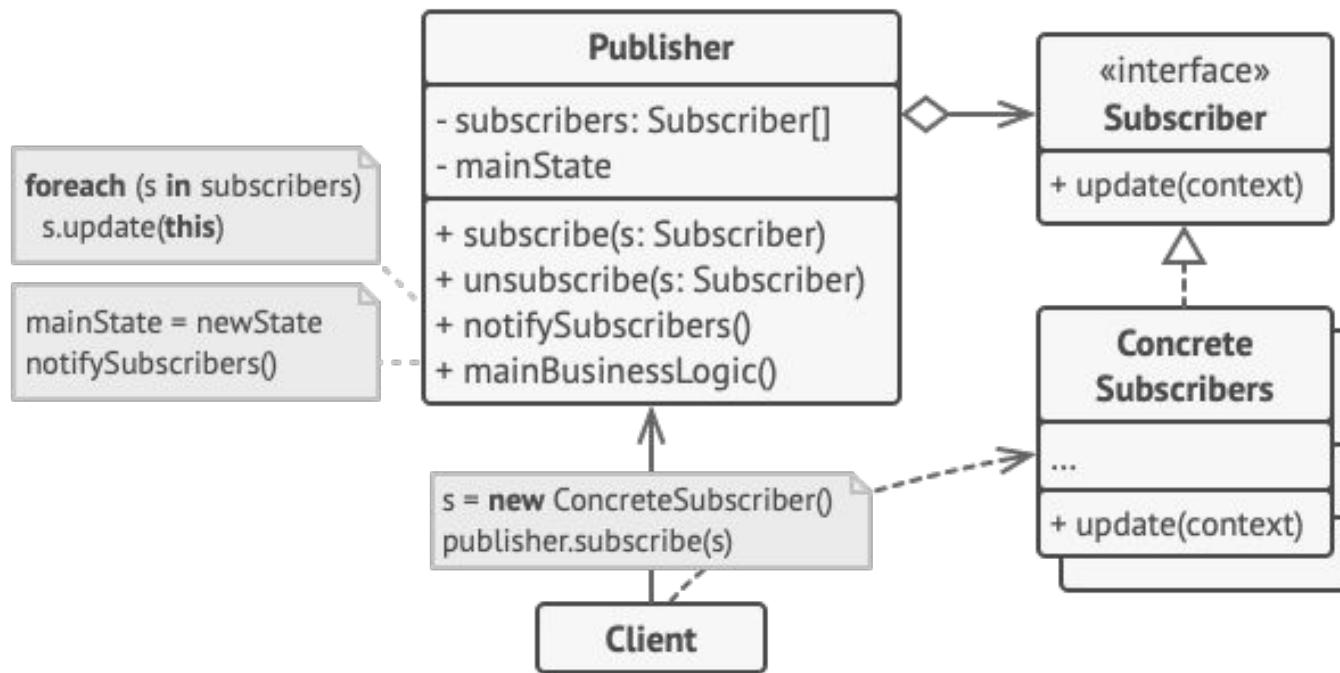
Game updates GUI?

- Let the Game tell the GUI that something happened



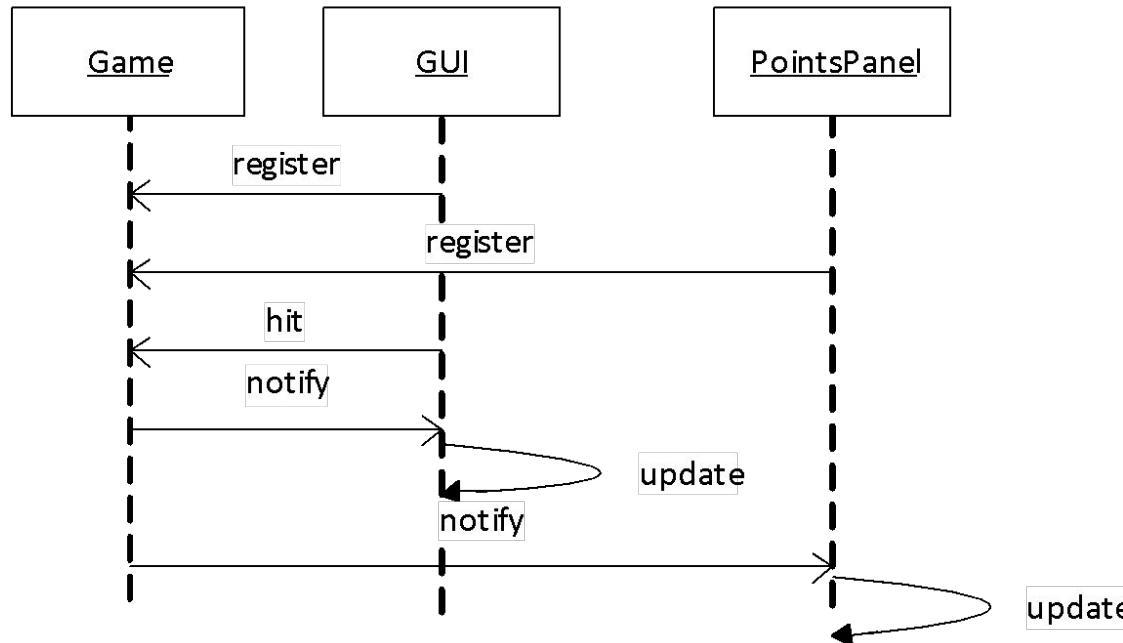
Problem: This couples the World to the GUI implementation.

Recall the Observer



Decoupling with the Observer pattern

- Let the Game tell *all* interested components about updates

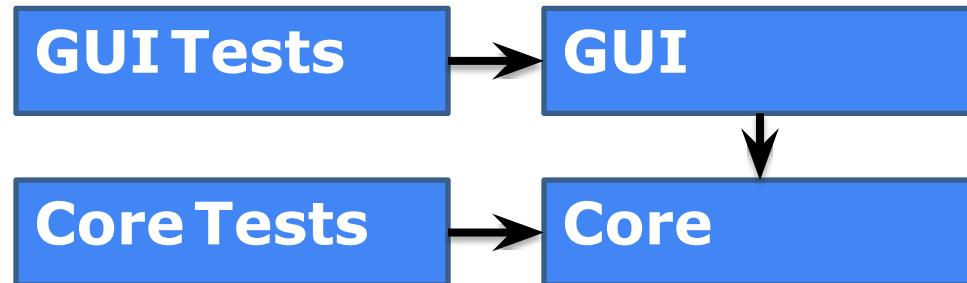


Core implementation vs. GUI

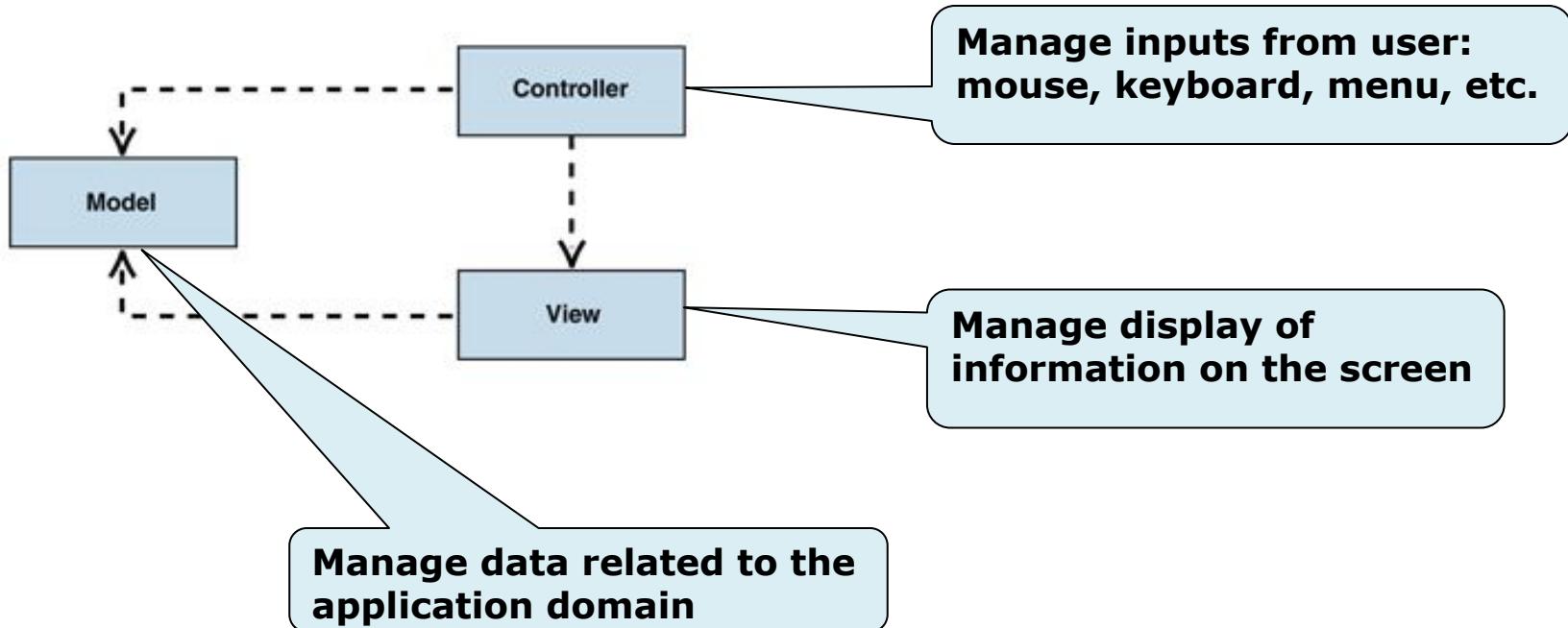
- Core implementation: application logic
 - Computing some result, updating data
- GUI
 - Graphical representation of data
 - Source of user interactions
- Design guideline: *avoid coupling the GUI with core application*
 - Multiple UIs with single core implementation
 - Test core without UI

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)

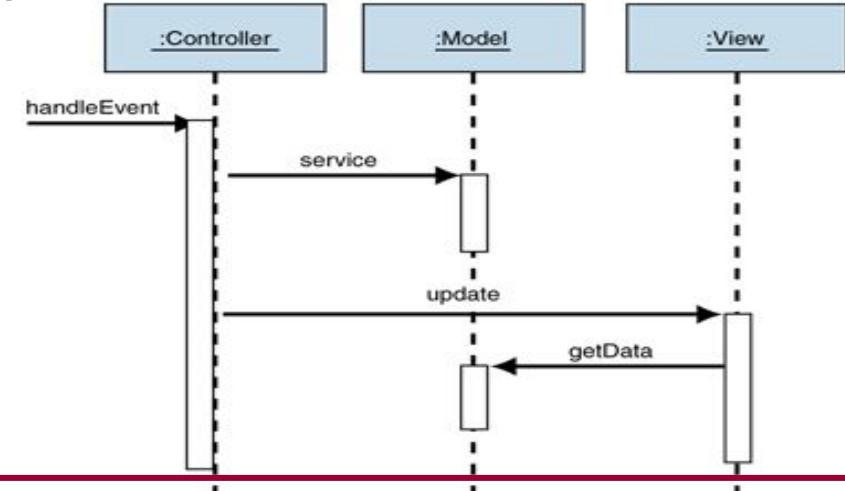
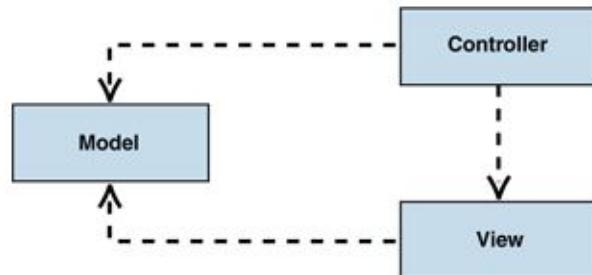


An architectural pattern: Model-View-Controller (MVC)

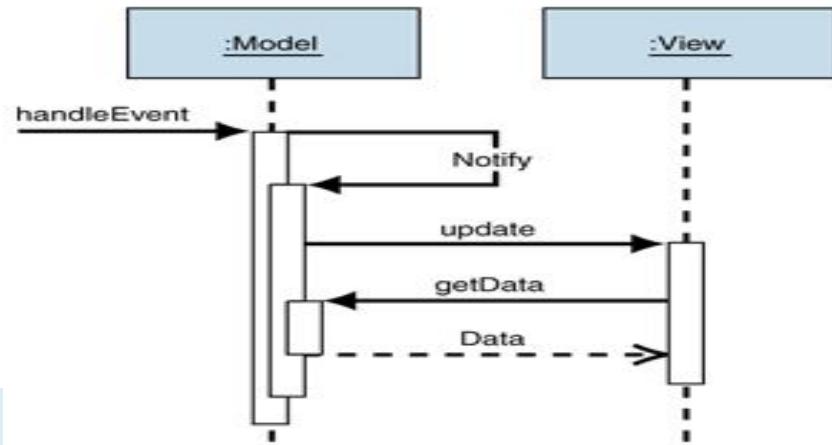
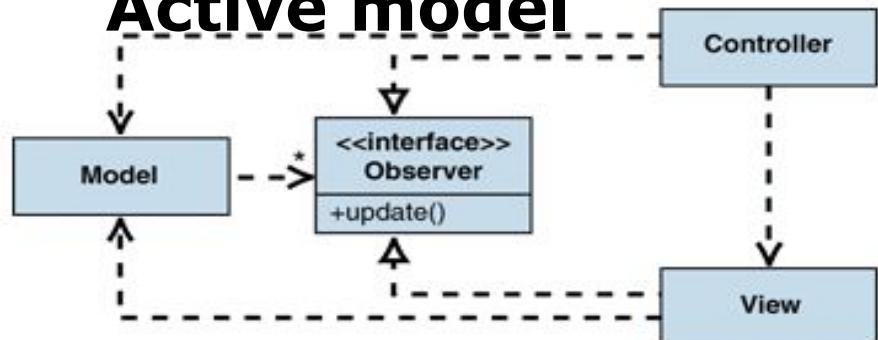


Model-View-Controller (MVC)

Passive model



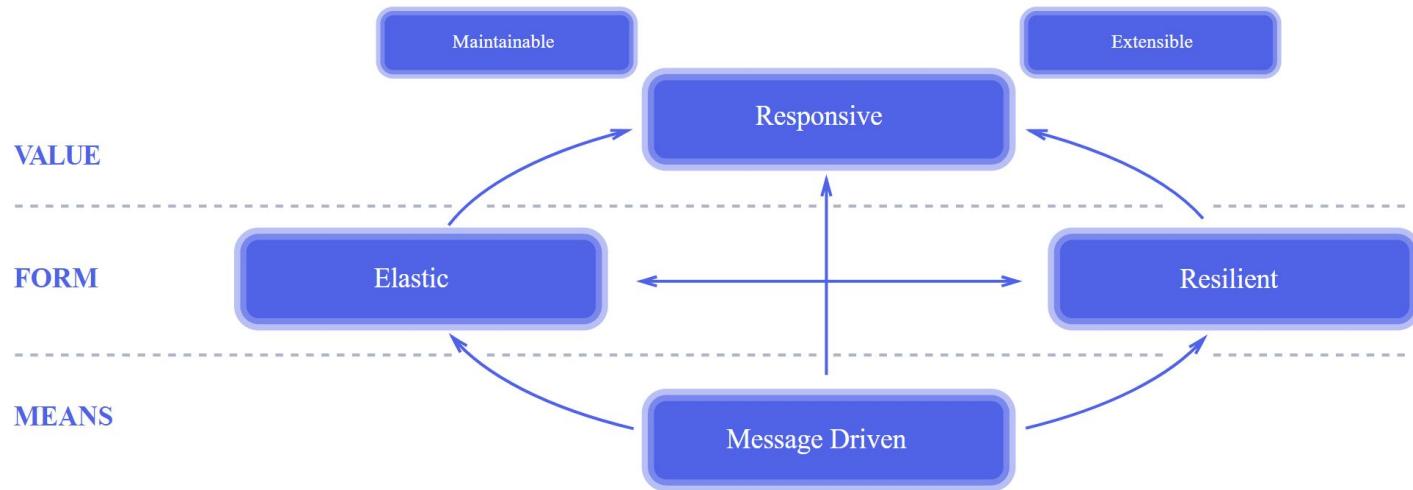
Active model



React Preview

How to handle asynchronous streams of data, across many actors?

- Without overwhelming workers
- Or blocking, or wasting resources



React Preview

“ReactiveX combines the **Observer pattern** with the **Iterator pattern** and *functional programming* with *collections* to fill the need for an ideal way of managing sequences of events.” <https://rxjs.dev/guide/overview>

“It extends the **observer pattern** to support sequences of data/events and adds operators that allow you to **compose** sequences together declaratively while abstracting away concerns about things like *low-level threading, synchronization, thread-safety and concurrent data structures.*” <https://github.com/ReactiveX/RxJava>

Summary

- Thinking past the main loop
 - The world is asynchronous
 - Concurrency helps, in a lot of ways
 - Requires revisiting programming patterns
- Start considering UI design
 - Discussed in more detail next week

Principles of Software Construction: Objects, Design, and Concurrency

Basic GUI concepts, HTML

Christian Kästner

Vincent Hellendoorn



HW3 Reflections

Today

- GUI Design
 - Concepts, strategies
 - Practical application in HTML, CSS, JS
- Dynamic Web Pages
 - Client/Server communication
 - Backend architecture

How To Make This Happen?

17-214 Fall 2021

Course calendar Schedule Syllabus Piazza

- Have experience testing and analyzing your software
- Understand principles of concurrency and distributed systems

See a more detailed list of [learning goals](#) describing what we want students to know or be able to do by the end of the semester. We evaluate whether learning goals have been achieved through assignments and exams.

Coordinates

Tu/Th 11:50 - 1:10 p.m. in DH 2315

Christian Kaestner, kaestner@cs.cmu.edu, TCS 345, office hours Friday 11:30-1pm (see calendar)

Vincent Hellendoorn, TCS 320, office hours Tuesdays 9am-11am (see calendar)

Our TAs also provide an additional 18h of office hours each week, usually in TCS 310, see details in the calendar.

The instructors have an open door policy: If the instructors' office doors are open and no-one else is meeting with us, we are happy to answer any course-related questions. Also, feel free to email us for appointments.

Course Calendar

17214 F21

Today October 2021

Sun	Mon	Tue	Wed	Thu	Fri	Sat
26	27	28	29	30	1 Oct	2
		14:00 Ye OH (Online) 15:00 Kevin OH	09:00 Vincent OH 11:50 Lecture	09:05 Recitation A (J 11:50 Midterm 10:10 Recitation B (T 14:00 Canceled: Oliv 11:15 Recitation C (J 17:00 Canceled: Kat +4 more	11:30 Christian OH 16:00 Zhifeng OH	10:00 Esther OH
3	4	5	6	7	8	9
	14:00 Ye OH (Online) 15:00 Kevin OH	09:00 Vincent OH 11:50 Lecture	09:05 Recitation A (J 10:00 Sophie OH 10:10 Recitation B (T 11:50 Lecture 11:15 Recitation C (J 14:00 Olivia OH +4 more 17:00 Katrina OH	16:00 Zhifeng OH	10:00 Esther OH	
10	11	12	13	14	15	16
	14:00 Ye OH (Online) 13:30 Christian OH 15:00 Kevin OH	09:00 Vincent OH 11:50 Lecture	09:05 Recitation A (J Midsemester Break 10:10 Recitation B (T 10:00 Sophie OH 11:15 Recitation C (J 14:00 Olivia OH +4 more 17:00 Katrina OH	16:00 Zhifeng OH	10:00 Esther OH	
17	18	19	20	21	22	23

Why not plaintext?

```
17-214 Fall 2021 Untitled-1 •  
1 17-214 Fall 2021  
2 Principles of Software Construction  
3 Objects, Design, and Concurrency  
4 Overview  
5 Software engineers today are less likely to design data structures and algorithms from scratch and more likely to build systems from library and framework components. In this course, students engage with concepts related to the construction of software systems at scale, building on their understanding of the basic building blocks of data structures, algorithms, program structures, and computer structures. The course covers technical topics in four areas: (1) concepts of design for complex systems, (2) object oriented programming, (3) static and dynamic analysis for programs, and (4) concurrent and distributed software. Student assignments involve engagement with complex software such as distributed massively multi-player game systems and frameworks for graphical user interaction.  
6  
7 Update for Fall 2021: We are planning several changes to the course for the fall 2021 semester. A key change is that we will teach the course with multiple programming languages. We will cover multiple languages in the lecture, but will expect students to focus on one language in assignments. When signing up, please chose a section for Java or JavaScript/TypeScript.  
8  
9 After completing this course, students will:  
10  
11 Be comfortable with object-oriented concepts and with programming in the Java or JavaScript language  
12 Have experience designing medium-scale systems with patterns  
13 Have experience testing and analyzing your software  
14 Understand principles of concurrency and distributed systems  
15 See a more detailed list of learning goals describing what we want students to know or be able to do by the end of the semester. We evaluate whether learning goals have been achieved through assignments and exams.  
16  
17 Coordinates  
18 Tu/Th 11:50 - 1:10 p.m. in DH 2315  
19  
20 Christian Kaestner, kaestner@cs.cmu.edu, TCS 345, office hours Friday 11:30-1pm (see calendar)  
21  
22 Vincent Heilendoorn, TCS 320, office hours Tuesdays 9am-11am (see calendar)  
23  
24 Our TAs also provide an additional 18h of office hours each week, usually in TCS 310, see details in the calendar.  
25  
26 The instructors have an open door policy: If the instructors' office doors are open and no-one else is meeting with us, we are happy to answer any course-related questions. Also, feel free to email us for appointments.  
27  
28 Course Calendar  
29  
30 Schedule  
31 We are planning significant changes to the course this semester. The schedule below is a rough draft of our plans, but likely to change.  
32  
33 TUE, AUG 31  
34 Intro  
35 WED, SEP 1  
36 rec 1 Introduction to Git  
37 THU, SEP 2  
38 OO basics, Dynamic dispatch, Encapsulation  
39 TUE, SEP 7  
40 IDEs, Build system, Continuous Integration, Libraries  
41 Required: Effective Java, Items 15 and 16  
42 WED, SEP 8  
43 rec 2 IDEs, Build systems, Libraries, CI  
44 THU, SEP 9  
45 Specifications and unit testing, exceptions
```

Why not a Doc?

17.asd: 10/11/2021 9:49 AM (Unsaved File) - Read-Only

Vincent Hellendoorn

Share Comments

Find Replace Select

Dictate Editor Reuse Files

RECOVERED UNSAVED FILE This is a recovered file that is temporarily stored on your computer. Save As

Navigation

Search document

Headings Pages Results

- Principles of Software Construction Objects, Design...
- Overview
- Coordinates
- Course Calendar

Schedule

- Course Syllabus and Policies
 - Prerequisites
 - Grading
 - Attendance and remote participation
 - Textbooks
 - Time management
 - Late work policy
 - Collaboration policy
 - Accommodations
 - Your health matters
- Informal feedback on this course

Have experience designing medium-scale systems with patterns
Have experience testing and analyzing your software
Understand principles of concurrency and distributed systems

See a more detailed list of [learning goals](#) describing what we want students to know or be able to do by the end of the semester. We evaluate whether learning goals have been achieved through assignments and exams.

Coordinates

TuTh 11:50 - 1:10 p.m. in DH 2315

[Christian Kaestner](#), kaestner@cs.cmu.edu, TCS 345, office hours Friday 11:30-1pm (see calendar)

[Vincent Hellendoorn](#), TCS 320, office hours Tuesdays 9am-11am (see calendar)

Our TAs also provide an additional 18h of office hours each week, usually in TCS 310, see details in the calendar.

The instructors have an [open door](#) policy. If the instructors' office doors are open and no-one else is meeting with us, we are happy to answer any course-related questions. Also, feel free to email us for appointments.

Course Calendar

7214 F21

Today October 2021

Sun Mon Tue Wed Thu Fri Sat

26 27 28 29 30 1 Oct 2

Print Week Month Agenda

14:00 Ye OH 15:00 Kevin OH 09:00 Vincent OH Recitation A 09:05 Midterm (Online only!) 11:30 Christian OH 16:00 Esther OH
11:30 Lecture OH 10:10 Recitation B 10:00 Canceled; Olivia OH - 15:00 Zhufeng OH (TypeScript)
- no office hours this week due to midterm grading
11:15 Recitation C 17:00 Canceled; Katrina OH

Page 2 of 21 3693 words

GUI Design: what do we want?

- Nested Elements
- Style Vocabulary
- Interactivity

GUI Design: what do we want?

- Nested Elements
 - HTML
- Style Vocabulary
 - CSS
- Interactivity
 - JavaScript

Anatomy of an HTML Page

Predefined elements

The diagram illustrates the structure of an HTML page. At the top, a box labeled "Root*" contains the text "Software Construction". Below it, a box labeled "Header" contains the text "Objects Design and Concurrency". A larger box labeled "Body" contains the text "Overview" and a detailed description of the course content. Three arrows point from these boxes to the corresponding HTML elements in a browser's developer tools. The "Root*" arrow points to the root element <!DOCTYPE html>. The "Header" arrow points to the <head> element. The "Body" arrow points to the <body> element, which is expanded to show its contents: <nav id="navigation" class="hidden">...</nav>, <header id="top" class="container">...</header>, <div id="main" class="container">...</div>, </body>, and </html>. The developer tools interface shows tabs for Elements, Console, Sources, Network, and other developer tools.

```
<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  ...<body> == $0
    <nav id="navigation" class="hidden">...</nav>
    <header id="top" class="container">...</header>
    <div id="main" class="container">...</div>
  </body>
</html>
```

html body

Styles Computed Layout Event Listeners DOM Breakpoints

Filter

element.style {
}
body {
 font-family: 'Helvetica Neue', Helvetica, sans-serif;
 font-size: .9em;
 padding: 50px 0;
}
body {

Anatomy of an HTML Page

Nested elements

- Sizing
- Attributes
- Text

The screenshot shows a browser window displaying a course page for "Principles of Software Construction". The page title is "Principles of Software Construction" and the subtitle is "Objects, Design, and Concurrency". Below the title, there is an "Overview" section with a paragraph of text about the course's focus on building systems from library and framework components. The browser's developer tools are open, specifically the "Elements" tab, which shows the DOM structure. The "body" element contains a "header" with id "top" and class "container", a "div" with id "main" and class "container", and an "h2" with id "overview" containing the text "Overview". The "main" div also contains a paragraph with a red color style and another paragraph with the text "After completing this course, students will:". The "Elements" tab has tabs for "Styles", "Computed", "Layout", "Event Listeners", and "DOM Breakpoints". At the bottom right, there is a detailed view of the element's bounding box with dimensions 355.200x14052.300 and padding of 50px.

17-214 Fall 2021

header#top.container 355.2 x 141.6

Principles of Software Construction

Objects, Design, and Concurrency

Overview

Software engineers today are less likely to design data structures and algorithms from scratch and more likely to build systems from library and framework components. In this course, students engage with concepts related to the construction of software systems at scale, building on their understanding of the basic building blocks of data structures, algorithms, program structures, and computer structures. The course covers technical topics in four areas: (1) concepts of design for complex systems, (2) object oriented programming, (3) static and dynamic analysis for

... ▾ <body> == \$0

▶ <nav id="navigation" class="hidden">...</nav>

▶ <header id="top" class="container">...</header>

▼ <div id="main" class="container">

 ::before

 <h2 id="overview">Overview</h2>

 ▶ <p>...</p>

 ▶ <p style="color: red">...</p>

 <p>After completing this course, students will:</p>

 ▶ ...

 ▶ <n> </n>

html body

Styles Computed Layout Event Listeners DOM Breakpoints

margin -

border -

padding 50

355.200x14052.300

50

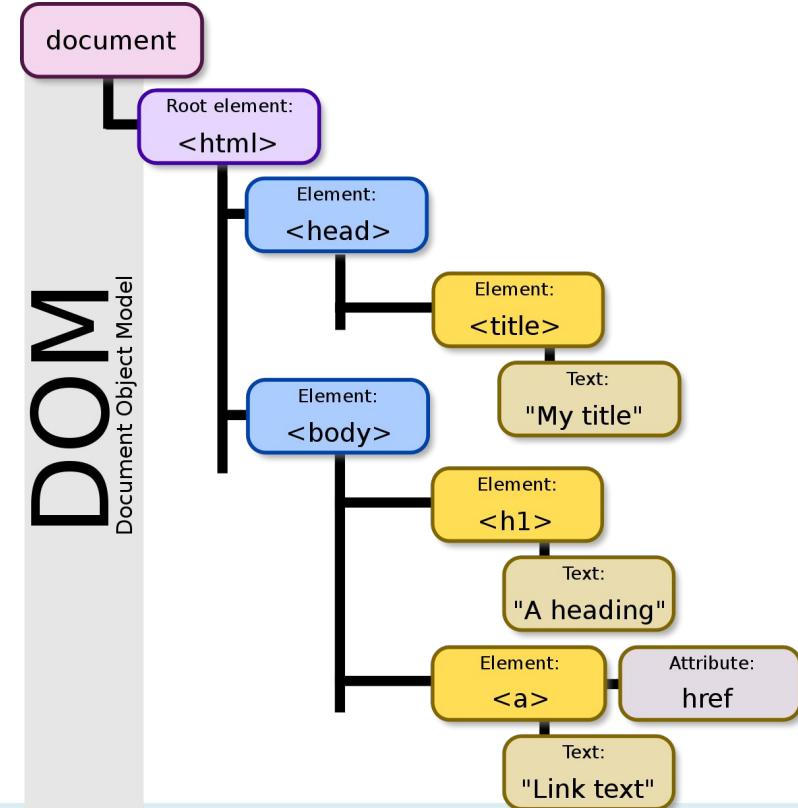
-

RESEARCH

Anatomy of an HTML Page

Many GUIs are trees

- Nested elements, recursively
- Some fixed positions (html, body)

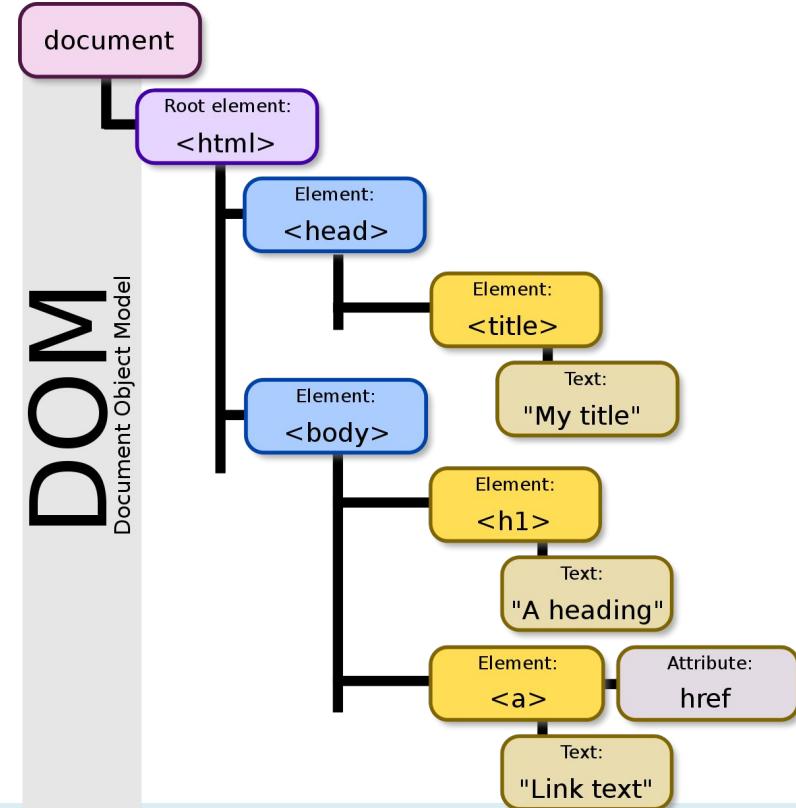


Anatomy of an HTML Page

Many GUIs are trees

- Nested elements, recursively
- Some fixed positions (html, body)

How to implement this?



The composite pattern

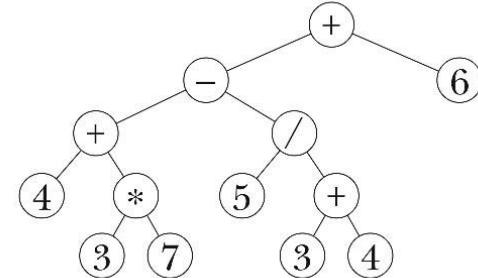
- Problem: Collection of objects has behavior similar to the individual objects
- Solution: Have collection of objects and individual objects implement the same interface
- Consequences:
 - Client code can treat collection as if it were an individual object
 - Easier to add new object types
 - Design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

Another composite pattern example

```
public interface Expression {  
    double eval();      // Returns value  
}
```

```
public class BinaryOperationExpression implements Expression {  
    public BinaryOperationExpression(BinaryOperator operator,  
                                     Expression operand1, Expression operand2);  
}
```

```
public class NumberExpression implements Expression {  
    public NumberExpression(double number);  
}
```



Composite

- Elements can contain elements
 - With restrictions
 - Need to deal with style, interaction
- In JS: HTMLElement
 - With child-classes e.g. HTMLDivElement, HTMLBodyElement
 - Navigation:
 - getElement*: locate by tag name, id, class, etc.
 - next/prev(Element)Sibling
 - childNodes, parent

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

Not only leaf-nodes have an appearance

The screenshot shows a browser window with the title "Course Calendar" and subtitle "17214 F21". Below this is a date navigation bar with "Today", left and right arrows, and the month "October 2021". The main area displays a calendar grid for October 2021. A tooltip indicates a table row has a height of 784 pixels and spans 18 columns. The calendar shows events like "14:00 Ye OH (Online)" and "15:00 Kevin OH" for various dates. The browser's developer tools are open, specifically the "Elements" tab, which shows the HTML DOM structure corresponding to the calendar. The DOM includes elements like "month-row", "st-bg-table", "st-grid", "tbody", "tr", and "td" with classes such as "st-dtitle", "st-dtitle-fc", "st-dtitle-today", and "st-dtitle-next". The "Elements" tab also lists other components like "mv-event-container", "div.month-row", "table.st-grid", "tbody", "tr", and "td.st-dtitle.st-dtitle-today".

Style

Tags come with inherent & customizable style

- Inherent:
 - <div> is a `block` (full-width, with margin)
 - is in-line
 - <h1> is large
- Customizable: add and override styles
 - Change font-styles, margins, widths
 - Modify groups of elements

Style: CSS

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

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

vs.

```
<style type="text/css">
    span {
        font-family: arial
    }
</style>
```

Style: CSS

- Cascading Style Sheets
 - Reuse: styling rules for tags, classes, types
 - Reuse: not just at the leafs!
- What if there are conflicts?

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

- Lowest element wins*

*Technically, there's a whole scoring system

Style: CSS

- Cascading Style Sheets
 - Reuse: styling rules for tags, classes, types
 - Reuse: not just at the leafs!
- What if there are no conflicts?

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

- How would you implement this?

Style: CSS

What is happening here?

The screenshot shows a browser's developer tools with the "Elements" tab selected. The left pane displays the page content: "Hi there!" in bold and "Hello again!" in regular weight. The right pane shows the DOM tree:

```
<span style="font-style:bold">Hi there!</span>
<br>
▼<div style="font-weight:normal">
...   <span style="font-weight:bold">Hello again!</span> == $0
</div>
</body>
</html>
```

Below the DOM tree, the "span" tab is selected in the "Styles" section. The "Properties" section shows the following CSS rules:

```
div > span {
    font-family: 'Times New Roman', Times, serif;
}

span {
    font-family: arial;
}

span {
    font-family: arial;
}
```

The first rule is from "main.css:13" and the second and third rules are from "index.html:6". The third rule is highlighted with a red border.

Decorator

What is happening here?

- To compute the style of an element:
 - Apply its tag-default style
 - **Wrap** in added style rules (tag-specific or general)
 - Text: font-family, weight, etc.
 - Inherit parents' style
 - Conflicts lead to overrides
- Makes *themes* really powerful

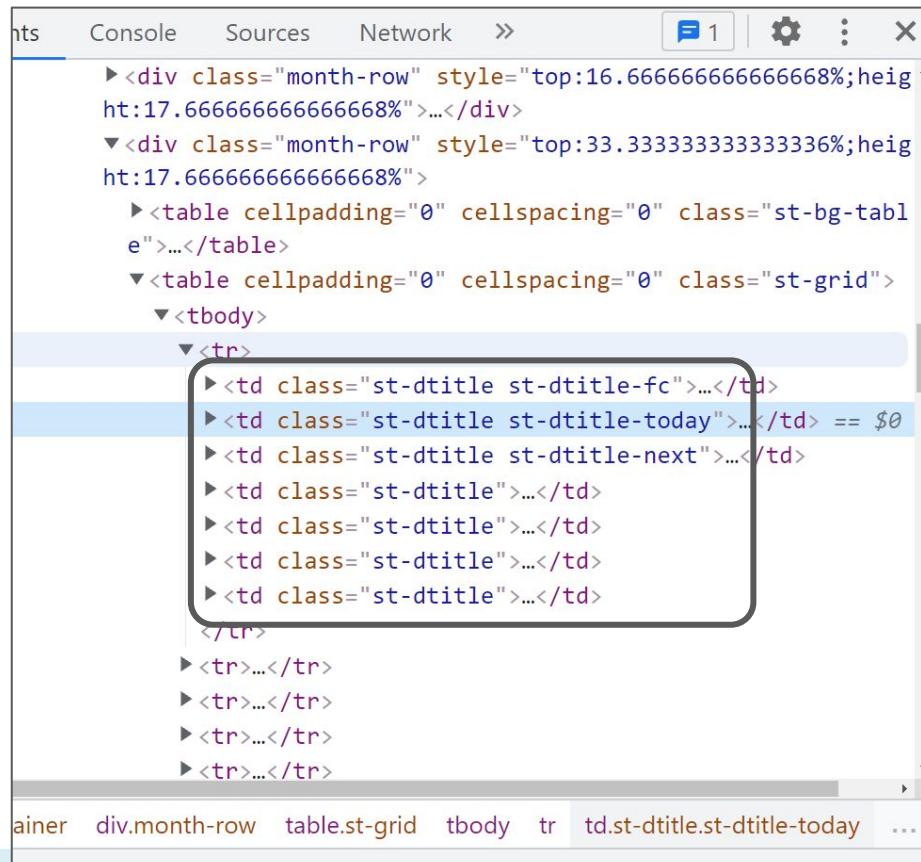
Technically, HTML is streamed top-to-bottom; CSS works bottom-up

CSS: classes

Let's not repeat custom style

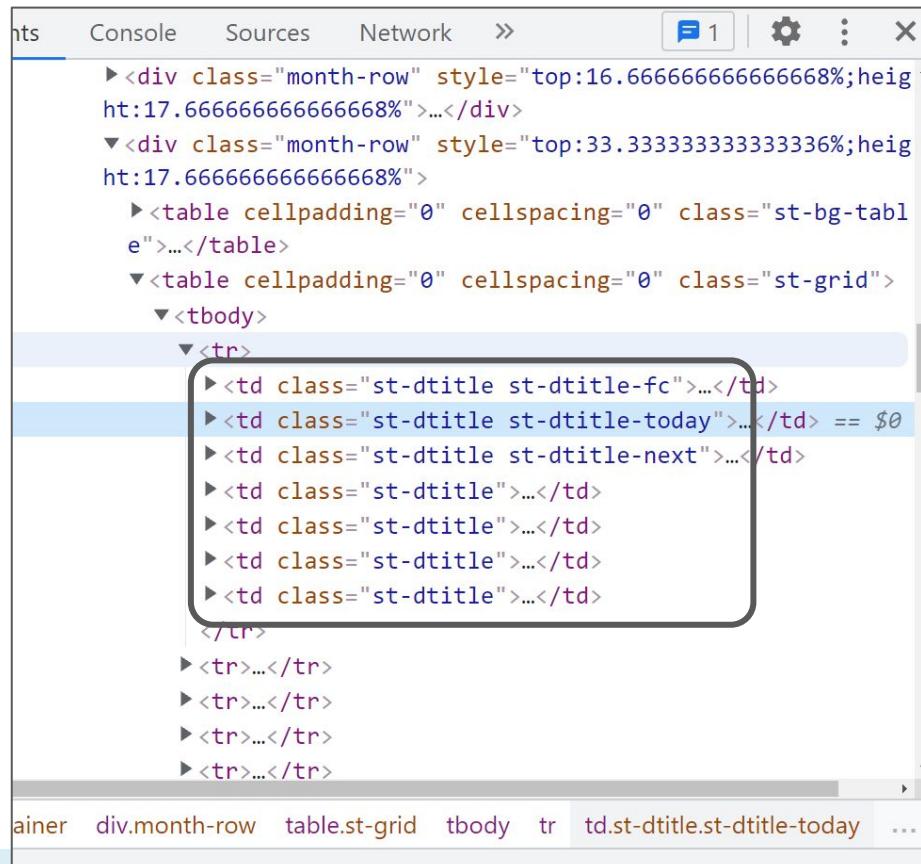
- Use any nr. of class label(s)
- Class styles get added
- Facilitates reuse

How would you implement this?



The screenshot shows the DOM tab in a browser developer tools window. It displays the HTML structure of a table row (`<tr>`). One specific `<td>` element is highlighted with a red rounded rectangle. This element has the class `st-dtitle st-dtitle-today`. The DOM tree also includes other `<td>` elements with classes like `st-dtitle-st-dtitle-fc`, `st-dtitle-st-dtitle-next`, and `st-dtitle`. The entire row (`<tr>`) and its closing tag (`</tr>`) are visible at the bottom of the tree.

Strategy or Observer?



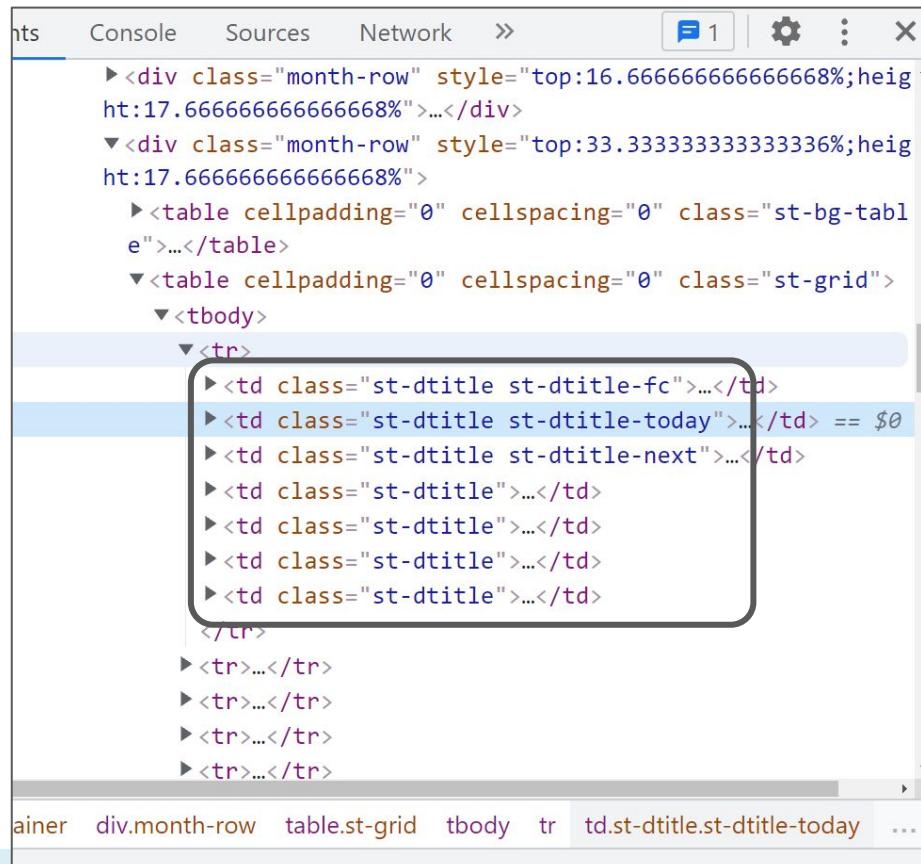
The screenshot shows a browser's developer tools DOM panel. The 'Elements' tab is selected, displaying a hierarchical tree of HTML elements. A specific `<td>` element is highlighted with a red rounded rectangle. This element is part of a `<tr>` row, which is itself part of a `<tbody>` section of a `<table>`. The `<table>` has the class `st-grid`. The entire row `<tr>` is also highlighted with a blue background.

```
> <div class="month-row" style="top:16.66666666666668%;height:17.66666666666668%">...</div>
  <div class="month-row" style="top:33.33333333333336%;height:17.66666666666668%">
    <table cellpadding="0" cellspacing="0" class="st-bg-table">...
      <table cellpadding="0" cellspacing="0" class="st-grid">
        <tbody>
          <tr>
            <td class="st-dtitle st-dtitle-fc">...</td>
            <td class="st-dtitle st-dtitle-today" style="background-color: #0070C0; color: white;">$0</td>
            <td class="st-dtitle st-dtitle-next">...</td>
            <td class="st-dtitle">...</td>
            <td class="st-dtitle">...</td>
            <td class="st-dtitle">...</td>
            <td class="st-dtitle">...</td>
          </tr>
          <tr>...</tr>
          <tr>...</tr>
          <tr>...</tr>
          <tr>...</tr>
        </tbody>
      </table>
    </div>
  </div>
```

Strategy or Observer?

Either could apply

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



The screenshot shows a browser's developer tools Network tab with several month-row elements listed. One specific td element is highlighted with a red rounded rectangle.

```
> <div class="month-row" style="top:16.66666666666668%;height:17.66666666666668%">...</div>
  <div class="month-row" style="top:33.33333333333336%;height:17.66666666666668%">
    <table cellpadding="0" cellspacing="0" class="st-bg-table">...
      <table cellpadding="0" cellspacing="0" class="st-grid">
        <tbody>
          <tr>
            <td class="st-dtitle st-dtitle-fc">...</td>
            <td class="st-dtitle st-dtitle-today">...</td> == $0
            <td class="st-dtitle st-dtitle-next">...</td>
            <td class="st-dtitle">...</td>
            <td class="st-dtitle">...</td>
            <td class="st-dtitle">...</td>
            <td class="st-dtitle">...</td>
          </tr>
        <tr>...</tr>
        <tr>...</tr>
        <tr>...</tr>
        <tr>...</tr>
```

ainer div.month-row table.st-grid tbody tr td.st-dtitle.st-dtitle-today ...

Interactivity

A GUI is more than a document

- How do we make it “work”?

The screenshot shows a browser's developer tools open to the 'Elements' tab. The page content includes the text "Hi there!" and "Hello again!", with "Hello again!" in bold. A button labeled "Click me" is also present. The DOM tree in the tools shows the following structure:

```
<br>
▼<div style="font-weight:normal">
...   <span style="font-weight:bold">Hello again!</span> == $0
</div>
<button>Click me</button>
</body>
</html>
```

The 'span' node under the first 'div' is highlighted. The bottom navigation bar of the tools has tabs for 'Styles', 'Computed', 'Layout', 'Event Listeners', 'DOM Breakpoints', 'Properties', and 'Accessibility'. The 'Styles' tab is currently selected.

Actions: JavaScript

- Key: event listeners (what's that pattern?)
- (frontend) JS is highly event-driven
 - Respond to window `onLoad` event, content loads (e.g., ads)
 - Respond to clicks, moves



Observer Pattern

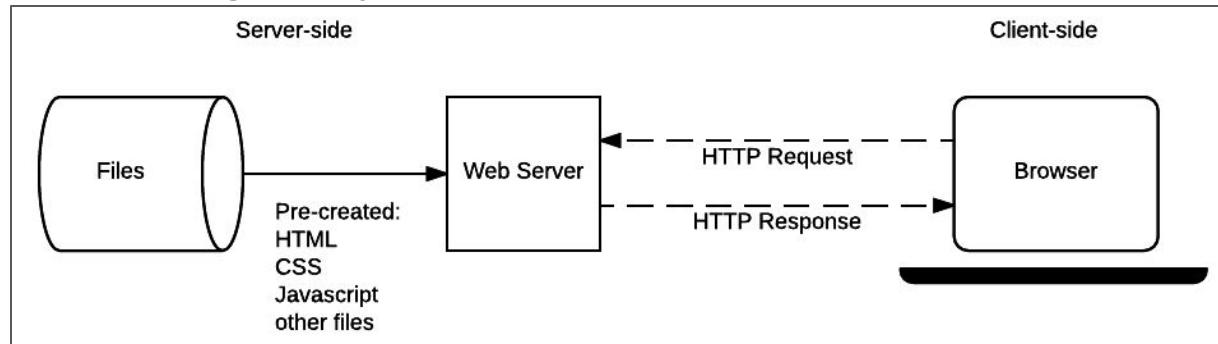
- Manages publishers and subscribers
 - Here, button publishes its 'click' events
 - `buttonClicked` subscribes to 1+ updates
- Flexibility and Reuse
 - Multiple observers per element
 - Shared observers across elements

Step Back

- What is our website now?
 - Layout, style, interaction
 - What is missing?

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



Static Web Pages

- Delivered as-is, final
 - Consistent, often fast
 - Cheap, only storage needed
- Maintain with *static website generators*
 - Or you'll be doing a lot of copying
 - Coupled with themes => rapid development, deployment
 - Quite popular, e.g. hosting on GH Pages

Static Web Pages

- But ...
 - No persistence (at least, not obviously)
 - No customizability (e.g., accounts)
 - No communication (payment, chat, etc)
 - Realistically, no intensive jobs

Dynamic Web Pages

- Client/Server
 - Someone needs to answer the website's calls
 - Doesn't need to be us!
 - Host a webserver
 - Serves pages, handles calls
 - For static pages too!
- We'll show you more tomorrow (Wednesday)

Web Servers

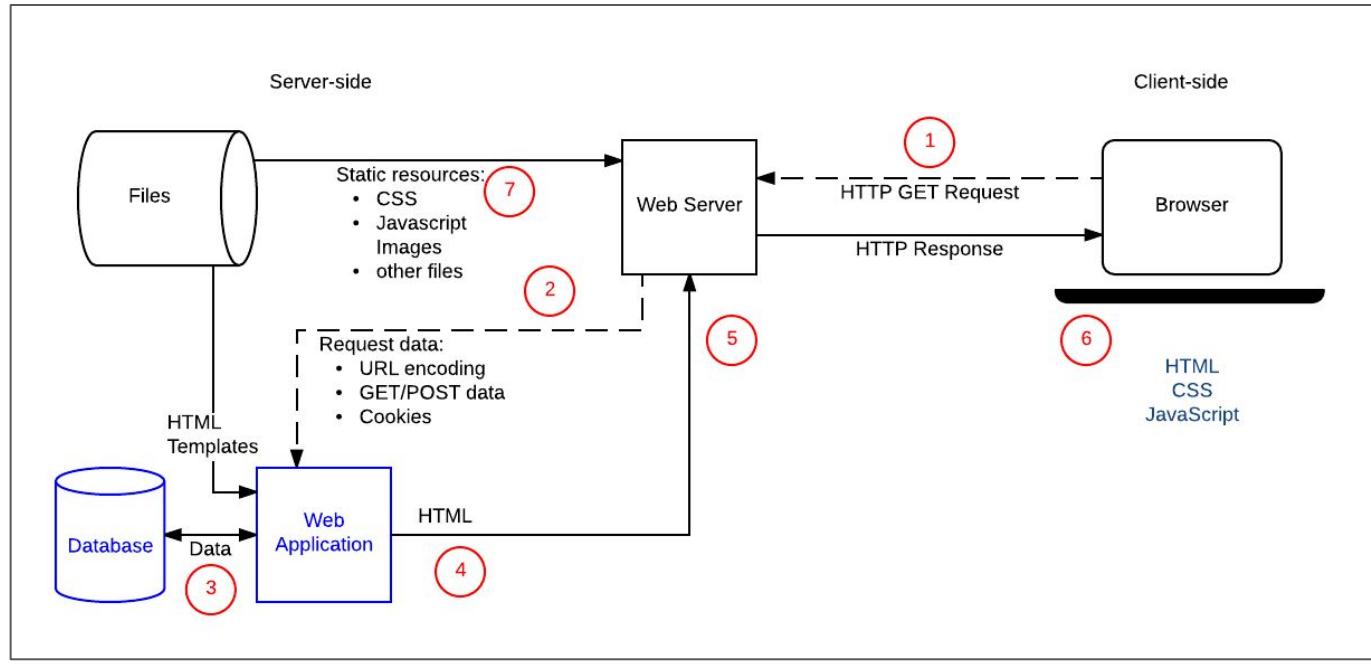
- Communicate via HyperText Transfer Protocol
 - URL (the address)
 - Method:
 - GET: retrieve data. Parameters in URL `...?key=value&key2=value2` and message body
 - POST: store/create data. Parameters in request body
 - Several more, rarely used
 - Responses:
 - *Status Code*. We all know 404. 2XX family is OK.
 - And possible data. E.g., entire HTML page.

Web Servers

- Communicate via HyperText Transfer Protocol
 - URL (the address)
 - Method:
 - GET: retrieve data. Parameters in URL `...?key=value&key2=value2` and message body
 - POST: store/create data. Parameters in request body
 - Several more, rarely used
 - Responses:
 - *Status Code*. We all know 404. 2XX family is OK.
 - And possible data. E.g., entire HTML page.
 - POST makes no sense for static sites!
 - As do GETs with parameters

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

AJAX

- Originally: “Asynchronous JavaScript and XML”
 - Updates parts of a page dynamically
 - Sends XMLHttpRequests with a callback
 - On return, check the code; handle success and failure.
 - Asynchronous, naturally decouples backend from UI

AJAX

- Originally: “Asynchronous JavaScript and XML”
 - Updates parts of a page dynamically
 - Sends XMLHttpRequests with a callback
 - On return, check the code; handle success and failure.
 - Asynchronous, naturally decouples backend from UI
- Slowly being phased out
 - Replace with `fetch`, which uses... Promises
 - More next week

How to Web App?

- Let's avoid generating HTML from scratch on every call
 - Map requests to handler code
 - Fetch data, process
 - Generate and return HTML
- Historically: PHP
 - Modifies HTML pages server-side on request; strong ties to SQL

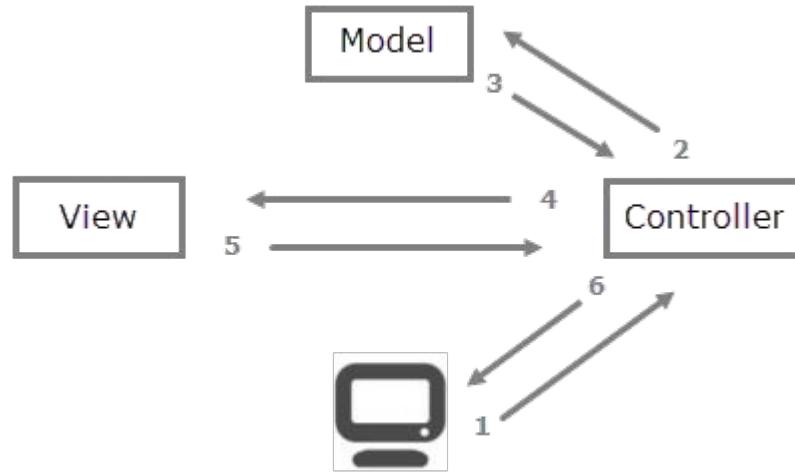
```
<?php
    // The global $_POST variable allows you to access the data sent with the POST method by name
    // To access the data sent with the GET method, you can use $_GET
    $say = htmlspecialchars($_POST['say']);
    $to  = htmlspecialchars($_POST['to']);

    echo $say, ' ', $to;
?>
```

How to Web App?

- Let's avoid generating HTML from scratch on every call
 - Map requests to handler code
 - Fetch data, process
 - Generate and return HTML
- Or use a framework
 - Python: Flask, Django
 - NodeJS: Express
 - Spring for Java
 - Many others, differences in **weight**, features

Model-View-Controller (MVC)



<https://overiq.com/django-1-10/mvc-pattern-and-django/>

MVC is ubiquitous

Separates:

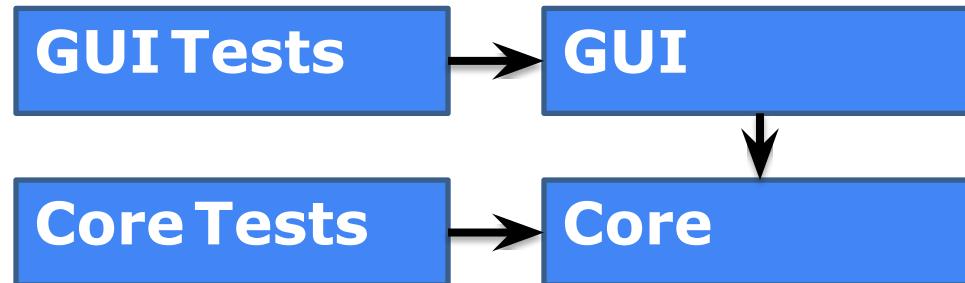
- Model: data organization
 - Interface to the database
- View: data representation (typically HTML)
 - Often called *templates* in web-dev; “view” is a bit overloaded
- Controller: intermediary between client and model/view
 - Typically asks model for data, view for HTML

Core implementation vs. GUI

- Core implementation: application logic
 - Computing some result, updating data
- GUI
 - Graphical representation of data
 - Source of user interactions
- Design guideline: *avoid coupling the GUI with core application*
 - Multiple UIs with single core implementation
 - Test core without UI

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)



Summary

- GUIs are full of design patterns
 - Helpful for reuse, delegation in complex environments
- Covered the basics of HTML, CSS, JS, servers
 - Needed for dynamic web pages
 - Decouple the GUI; architect your backend
 - A lot more to learn (security, performance, privacy), but this will do
- You will build this
 - At a small scale

Principles of Software Construction: Objects, Design, and Concurrency

Concurrency: Safety & Immutability

Christian Kästner

Vincent Hellendoorn



Today

- A bit more on GUIs
 - Why HTML?
 - Event Handling
- Concurrency Patterns
 - Immutability
 - Safety, liveness
 - Designing for Concurrency

Mini-Quiz

<https://rb.gy/heh2ks>



HTML: how did we get here?

- Up till Spring, this course leaned on Java Swing
 - Obviously not compatible with JS
 - But also, fading in support

Swing

Anyone know of an app using a Swing UI?

Components of a Swing application

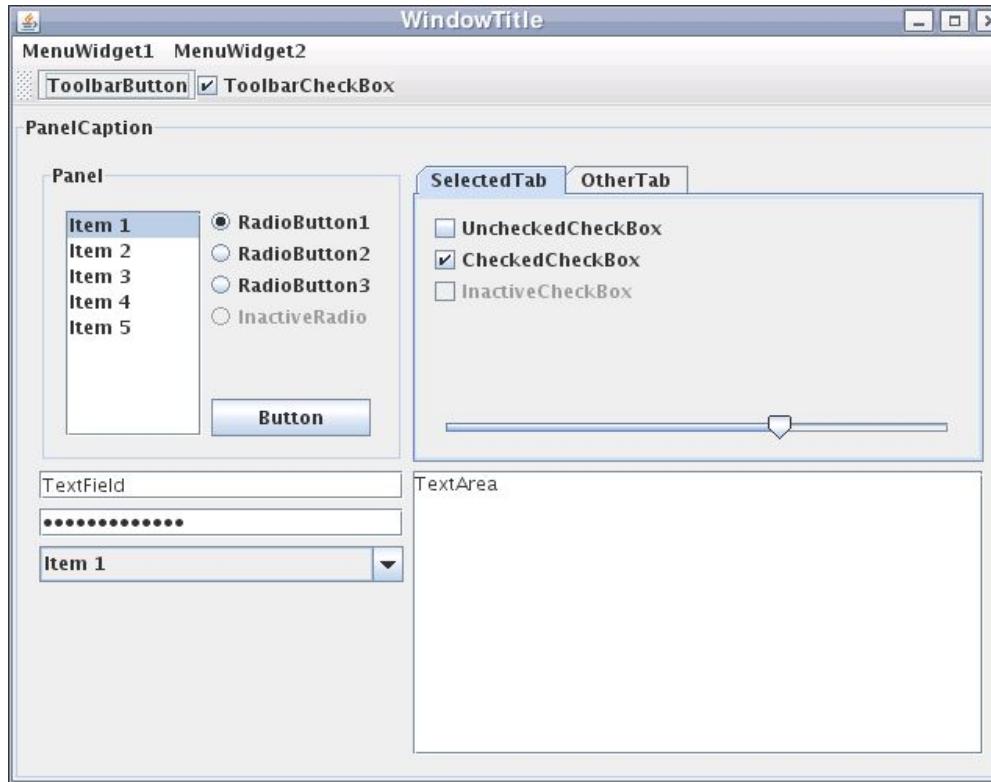
JFrame

JPanel

JButton

JTextField

...

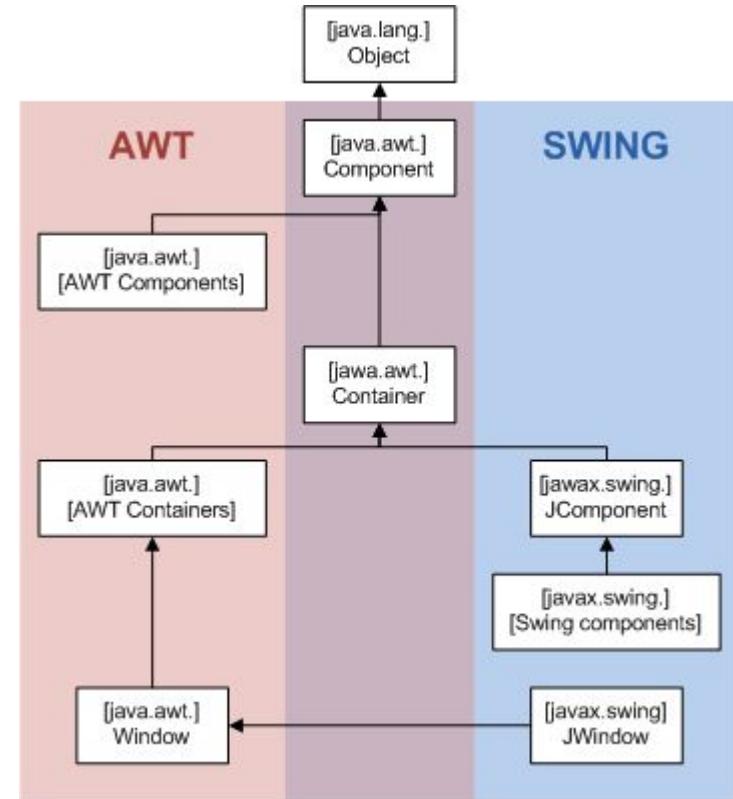


Quick Swing Demo

```
import javax.swing.*;  
  
public class SwingDemo extends JFrame {  
    private final JButton b = new JButton();  
  
    public SwingDemo() {  
        super();  
        this.setTitle("Swing Demo");  
        this.setBounds( x: 100, y: 100, width: 180, height: 140);  
        this.add(makeButton());  
        this.setVisible(true);  
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);  
    }  
  
    private JButton makeButton() {  
        b.setText("Click me!");  
        b.setBounds( x: 40, y: 40, width: 100, height: 30);  
        b.addActionListener(e -> JOptionPane.showMessageDialog(b, message: "Hello World!"));  
        return b;  
    }  
  
    public static void main(String[] args) throws InterruptedException, InvocationTargetException {  
        // Swing calls must be run by the event dispatching thread.  
        SwingUtilities.invokeLater(() -> new SwingDemo());  
    }  
}
```

So what is AWT doing here?

- Abstract Window Toolkit
 - The original Java UI
 - Wraps native code, so heavily platform-dependent



<https://en.wikipedia.org/wiki/File:AWTSwingClassHierarchy.png>

AWT

Why be platform-dependent?

Look and Feel

Eternal dilemma

- Platform-specific:
 - Better integration in terms of speed, appearance, features
- Platform-agnostic:
 - Broader deployment, more uniform experience
 - E.g., tablet, phone, computer, tv

Look and Feel

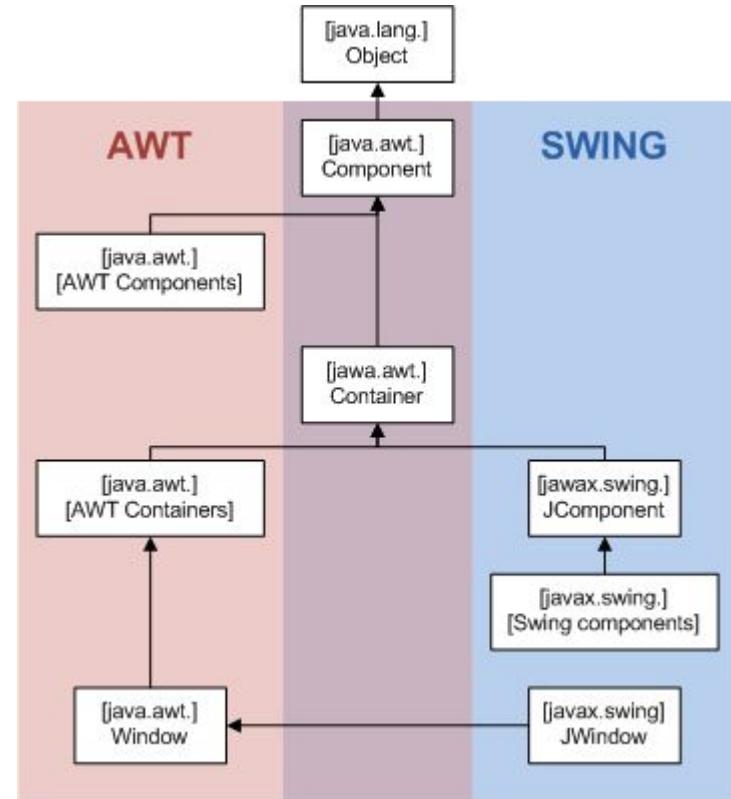
Eternal dilemma

- Platform-specific:
 - Better integration in terms of speed, appearance, features
- Platform-agnostic:
 - Broader deployment, more uniform experience
 - E.g., tablet, phone, computer, tv

Which one is HTML+CSS?

So what is AWT doing here?

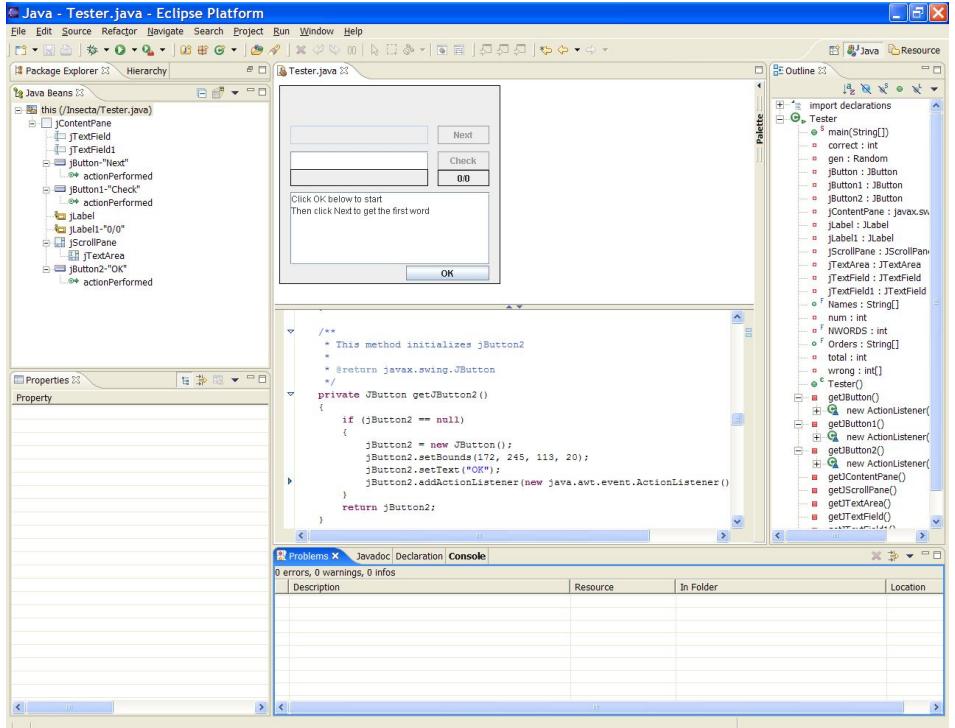
- To compare with Swing
 - Swing draws its own widgets
 - Using Java2D
 - Requires no native resources
- Swing still leans on AWT
 - So not quite “lightweight”



<https://en.wikipedia.org/wiki/File:AWTSwingClassHierarchy.png>

What about SWT?

- Powers Eclipse IDE
 - Developed by IBM
- Uses native code
 - Like AWT
 - But also provides own GUI code, when absent



https://en.wikipedia.org/wiki/Standard_Widget_Toolkit#/media/File:EclipseScreenshot.png

Which One is Better?

- Perhaps a matter of preference
 - Benchmarks show no real performance diff. between Swing & SWT
- Then there's Android, iOS, various wrappers (e.g., One UI)
- Why does this matter?

HTML + CSS

- Once upon a time, a web-page specific language

The screenshot shows a browser window with the developer tools open, specifically the Elements tab. The main content area displays a "Course Calendar" for "17214 F21" in "October 2021". The calendar shows events for Sunday, October 26, Monday, October 27, and Tuesday, October 28. The calendar uses a light purple background for the days and blue text for event details. A tooltip "tr 784 x 18" points to the table row structure. The developer tools sidebar shows the DOM tree, starting with a "month-row" div for each day, followed by a table with rows for each event. The "Elements" tab is selected, and the status bar at the bottom shows the path "... 2.mv-event-container div.month-row table.st-grid tbody tr td.st-dtitle.st-dtitle-today".

```
> <div class="month-row" style="top:16.66666666666668%;height:17.66666666666668%">...</div>
  <div class="month-row" style="top:33.33333333333336%;height:17.66666666666668%">
    <table cellpadding="0" cellspacing="0" class="st-bg-table">...</table>
    <table cellpadding="0" cellspacing="0" class="st-grid">
      <tbody>
        <tr>
          <td class="st-dtitle st-dtitle-fc">...</td>
          <td class="st-dtitle st-dtitle-today" style="background-color: #007bff; color: white;">= $0
```

HTML + CSS

- Grown into a general UI language
 - Involved some consolidation as recently as 2019
- Specifically, we are on HTML5
 - A “living standard”
 - Rich multimedia support, incl. SVG, video, audio, “canvas”



HTML + CSS

- Broadly adopted for GUI design
 - Including new settings, such as app development
 - E.g., with Cordova
 - Easy use with template engines
 - Like Handlebars



Today

- A bit more on GUIs
 - Why HTML?
 - **Event Handling**
- Concurrency Patterns
 - Immutability
 - Safety, liveness
 - Designing for Concurrency

Looping back to Event Loops

- Where are we “listening”?

```
private JButton makeButton() {  
    b.setText("Click me!");  
    b.setBounds( x: 40,  y: 40,  width: 100,  height: 30);  
    b.addActionListener(e -> JOptionPane.showMessageDialog(b,  message: "Hello World!"));  
    return b;  
}
```

There's a thread for that

- The **Event Dispatch Thread (EDT)**
 - Job: wait and dispatch
 - For JS, which is single-threaded, involve an **Event Loop** (later)

There's a thread for that

- The **Event Dispatch Thread (EDT)**
 - Job: wait and dispatch
 - For JS, which is single-threaded, involve an **Event Loop** (later)
- This thread is pretty busy
 - Move your mouse, hit keys? It's listening
 - For instance, Swing's EDT calls `actionPerformed` to notify subscribers
 - It needs to handle things quickly or the UI blocks
 - So don't waste its time!

There's a thread for that

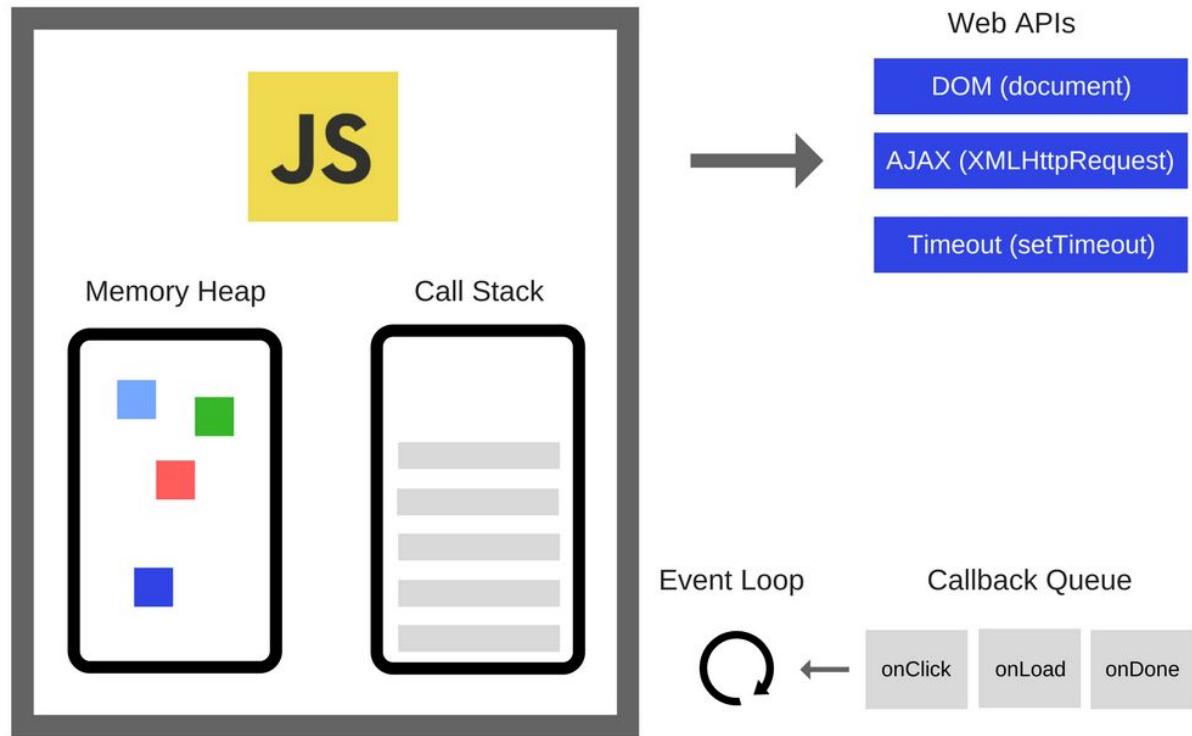
- This is why we `invokeAndWait`
 - Hand control of the task to Swing

```
public static void main(String[] args) throws InterruptedException, InvocationTargetException {  
    // Swing calls must be run by the event dispatching thread.  
    SwingUtilities.invokeAndWait(() -> new SwingDemo());  
}
```

Event Loop

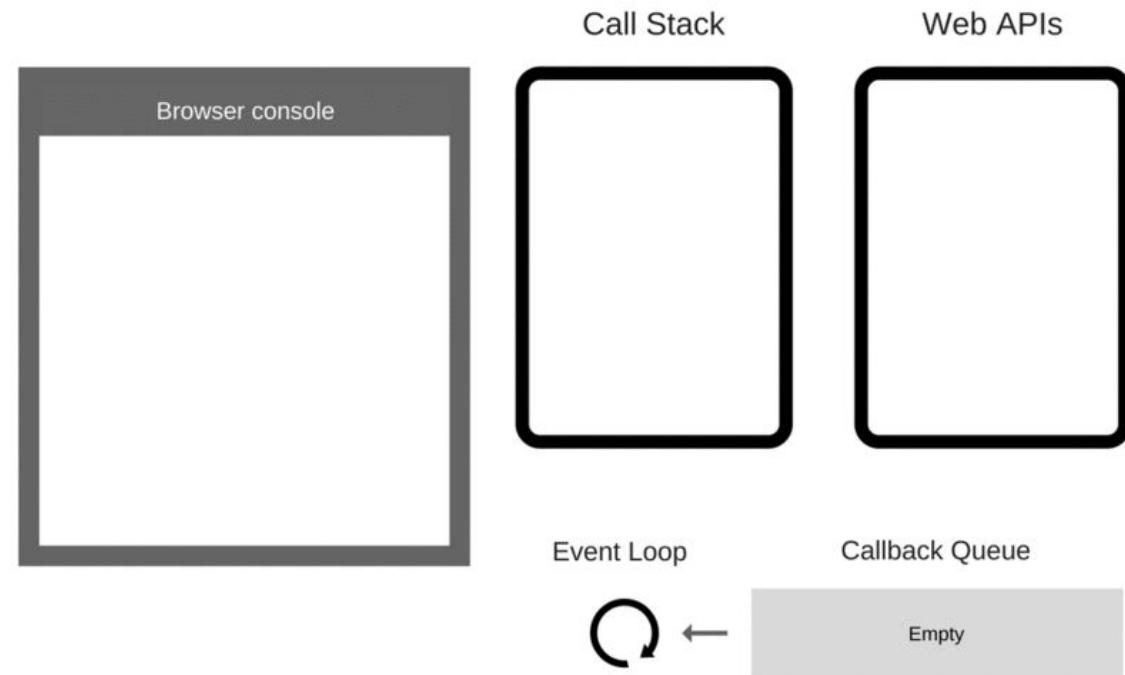
- At the heart, operates with a **queue**
 - Messages get added to the end
 - Oldest message are processed first
- In JS:
 - Waits *synchronously*
 - Executes each task *completely* without task-switching

Event Loop in JS



Event Loop in JS

1 / 16



Event Loops

- So JS never blocks
 - Meaning, the thread is never waiting to be granted power
 - (modulo rare exceptions)
 - Does that mean it is always responsive?

Event Loops

- So where do we do “heavy” work?

Event Loops

- So where do we do “heavy” work?
 - Chunk up slightly larger jobs
 - Allows other events to be handled in between
 - If we really need parallelism: WebWorkers
 - E.g., for rendering complex/large scenes
 - Ideally, move heavy work to the backend
 - A GUI shouldn’t be doing much work anyways

Event Loops

More on jobs and promises on Thursday

Forming Design Patterns

- We've seen:
 - Function-based dispatch (callbacks)
 - Using queues to manage asynchronous events
- Some of the building blocks of concurrent, distributed systems

Today

- A bit more on GUIs
 - Why HTML?
 - Event Handling
- **Concurrency Patterns**
 - Immutability
 - Safety, liveness
 - Designing for Concurrency

What if my Thread isn't Alone?

- Recall, in JS event loops:
 - Waiting is synchronous
 - Each message is processed fully without interruption
- What if we wanted multiple threads?
 - For parallelism
 - Multiple users on a website

What will Happen:

```
public class Synchronization {  
    static long balance1 = 100;  
    static long balance2 = 100;  
  
    public static void main(String[] args) throws InterruptedException {  
        Thread thread1 = new Thread(Synchronization::from1To2);  
        Thread thread2 = new Thread(Synchronization::from2To1);  
  
        thread1.start(); thread2.start();  
        thread1.join(); thread2.join();  
        System.out.println(balance1 + ", " + balance2);  
    }  
  
    private static void from1To2() {  
        for (int i = 0; i < 10000; i++) {  
            balance1 -= 100;  
            balance2 += 100;  
        }  
    }  
  
    private static void from2To1() {  
        for (int i = 0; i < 10000; i++) {  
            balance2 -= 100;  
            balance1 += 100;  
        }  
    }  
}
```

What will Happen:

Where does this fail?

What if single threaded?

Could we make it work
with 2 threads?

```
public class Synchronization {  
    static long balance1 = 100;  
    static long balance2 = 100;  
  
    public static void main(String[] args) throws InterruptedException {  
        Thread thread1 = new Thread(Synchronization::from1To2);  
        Thread thread2 = new Thread(Synchronization::from2To1);  
  
        thread1.start(); thread2.start();  
        thread1.join(); thread2.join();  
        System.out.println(balance1 + ", " + balance2);  
    }  
  
    private static void from1To2() {  
        for (int i = 0; i < 10000; i++) {  
            balance1 -= 100;  
            balance2 += 100;  
        }  
    }  
  
    private static void from2To1() {  
        for (int i = 0; i < 10000; i++) {  
            balance2 -= 100;  
            balance1 += 100;  
        }  
    }  
}
```

Atomicity

Competing access needs
to be managed.

```
public class Synchronization {  
    static AtomicInteger balance1 = new AtomicInteger( initialValue: 100);  
    static AtomicInteger balance2 = new AtomicInteger( initialValue: 100);  
  
    public static void main(String[] args) throws InterruptedException {  
        Thread thread1 = new Thread(Synchronization::from1To2);  
        Thread thread2 = new Thread(Synchronization::from2To1);  
  
        thread1.start(); thread2.start();  
        thread1.join(); thread2.join();  
        System.out.println(balance1 + ", " + balance2);  
    }  
  
    private static void from1To2() {  
        for (int i = 0; i < 10000; i++) {  
            balance1.getAndAdd( delta: -100);  
            balance2.getAndAdd( delta: 100);  
        }  
    }  
  
    private static void from2To1() {  
        for (int i = 0; i < 10000; i++) {  
            balance1.getAndAdd( delta: 100);  
            balance2.getAndAdd( delta: -100);  
        }  
    }  
}
```

Atomicity

Competing access needs to be managed.

- Atomic operations take place as a single unit
 - `getAndAdd` == read and write -- nobody else gets to touch it.
 - Is `balance++` atomic?
 - How about `pauseThread = true`

How to Prevent Competing Access?

- Any other ideas?

How to Prevent Competing Access?

- Any other ideas?
 - Don't have state!
 - Don't have shared state!
 - Don't have shared mutable state!

Today

- A bit more on GUIs
 - Why HTML?
 - Event Handling
- **Concurrency Patterns**
 - **Immutability**
 - Safety, liveness
 - Designing for Concurrency

Immutability

- A key principle in design, not just for concurrency
 - Inherently Thread-safe
 - No risks in sharing
 - Can make things very simple

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

Immutability

What if you need to make a change?

Immutability

What if you need to make a change?

```
function newGame(board: Board, nextPlayer: Player, history: Game[]): Game {
    return {
        board: board,
        play: function (x: number, y: number): Game {
            if (board.getCell(x,y)!==null) return this
            if (this.getWinner()!==null) return this
            const newHistory = history.slice()
            newHistory.push(this)
            return newGame(
                board.updateCell(x, y, nextPlayer),
                1 - nextPlayer,
                newHistory)
        }
    }
}, https://github.com/CMU-17-214/rec07-gui/blob/7e9f9202f22d3e015a1f7dd422794834f3386d4d/ts-express/src/game.js
```

Immutability

What functionality was made really easy by this design?

```
function newGame(board: Board, nextPlayer: Player, history: Game[]): Game {  
    return {  
        board: board,  
        play: function (x: number, y: number): Game {  
            if (board.getCell(x,y)!==null) return this  
            if (this.getWinner()!==null) return this  
            const newHistory = history.slice()  
            newHistory.push(this)  
            return newGame(  
                board.updateCell(x, y, nextPlayer),  
                1 - nextPlayer,  
                newHistory)  
        },  
    }  
}
```

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);  
    }  
}
```

Immutability

Any disadvantages?

Immutability

Any disadvantages?

```
String x = "It was the best of times, .."; // An entire book.  
x += "The end.";
```

Immutability

Any disadvantages?

```
String x = "It was the best of times, .."; // An entire book.  
x += "The end.";
```

- Provide mutable helpers (e.g. StringBuilder).
- Bundle common actions

Designing for Immutability

In short: make things immutable unless you really can't

- Especially, smaller data-classes
- Not realistic for classes whose state naturally changes
 - BankAccount: return a new account for each transaction?
 - In that case, minimize mutable part

Today

- A bit more on GUIs
 - Why HTML?
 - Event Handling
- **Concurrency Patterns**
 - Immutability
 - **Safety, liveness**
 - Designing for Concurrency

Thread Safety

- Let's define what we want:
 - **Thread safe** means no assumptions required to operate correctly with multiple threads.
 - Why was the earlier example not thread-safe?

Thread Safety

- Let's define what we want:
 - **Thread safe** means no assumptions required to operate correctly with multiple threads.
 - Why was the earlier example not thread-safe?
- If a program is not thread-safe, it can:
 - Corrupt program state (as before)
 - Fail to properly share state (cause liveness failure)
 - Get stuck in infinite mutual waiting loop (deadlock)

Back to: Atomicity

- Recall: atomic operations take place as a single unit
 - Read and write -- nobody else gets to touch it.
- Is atomicity sufficient for thread-safety?

Liveness Failure

```
public class LivenessFailure {  
  
    private static boolean stopRequested;  
  
    public static void main(String[] args) throws InterruptedException {  
        Thread backgroundThread = new Thread(new Runnable() {  
            public void run() {  
                int i = 0;  
                while (!stopRequested) {  
                    i++;  
                }  
            }  
        });  
        backgroundThread.start();  
        TimeUnit.SECONDS.sleep( timeout: 1);  
        stopRequested = true;  
    }  
}
```

Back to: Atomicity

- Recall: atomic operations take place as a single unit
 - Read and write -- nobody else gets to touch it.
- Is atomicity sufficient for thread-safety?
 - No. Shared memory is complicated

Shared State

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

```
public class VolatileExample {  
    private static volatile long nextSerialNumber = 0;  
  
    public static long generateSerialNumber() {  
        return nextSerialNumber++;  
    }  
  
    public static void main(String[] args) throws InterruptedException {  
        Thread threads[] = new Thread[5];  
        for (int i = 0; i < threads.length; i++) {  
            threads[i] = new Thread(() -> {  
                for (int j = 0; j < 1_000_000; j++)  
                    generateSerialNumber();  
            });  
            threads[i].start();  
        }  
        for(Thread thread : threads)  
            thread.join();  
        System.out.println(generateSerialNumber());  
    }  
}
```

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?

Synchronization

- Safe Communication + Exclusion
 - Requires a lock. In Java, tied to an object instance.
 - Complete ownership of resource, no caching risks.
 - Can make parallelism quite slow!



Back to “Blocking”

- Why does JS not have these issues?
 - Atomicity? Shared Reality? Safety?

Back to “Blocking”

- Why does JS not have these issues?
 - Atomicity: no thread can interrupt an action
 - The event loop completely finishes each task
 - Shared reality: no concurrent reads possible
 - Single-threaded by design
 - Safety: obvious.
- But, more burden on developers!

Is Threading all Bad?

- Not at all!
 - Obviously useful for parallelism and asynchronous I/O
 - But also, we can have **good design**.
- Threads map to tasks
 - Commonly assign one thread per task
 - Convenient abstract for handling large workloads
- Help manage complex event loops
 - Message passed from one handle to another in single-threaded envs.
 - See ‘promises’ on Thursday

Synchronization

There is a lot more to discuss

- How to synchronize, avoid deadlocks
- Active vs. passive waiting

Today

- A bit more on GUIs
 - Why HTML?
 - Event Handling
- Concurrency Patterns
 - Immutability
 - Safety, liveness
 - **Designing for Concurrency**

Forming Design Patterns

- We've seen:

Concurrency strategies:

- Function-based dispatch (callbacks)
- Using queues to manage asynchronous events

Thread-safety strategies:

- Immutability where possible
- Synchronization on mutable state

Forming Design Patterns

- We've not yet talked about:
 - Handling complex/multiple callbacks
 - Promises, Async/await
 - Guarding entire objects
 - Concurrency Encapsulation
 - Managing consumers & producers
 - Coupling, performance

Designing with Concurrency in Mind

- More on Thursday

Summary

- Event Loops require a different attitude
 - Avoid heavy lifting; think about blocking
 - More on Thursday
- Concurrency comes with some head-aches
 - Shared state is very complicated. Avoid it entirely!
 - Or synchronize well -- steep learning curve.
- Thursday:
 - “Callback hell” and why we need promises
 - Bits on React

HW4 Effort

<https://rb.gy/qyjpof>



Principles of Software Construction: Objects, Design, and Concurrency

Concurrency: Patterns & Promises

Christian Kästner

Vincent Hellendoorn



Today

- Design for Concurrency
 - How to: design for extension, reuse, readability, robustness?
 - The promise (future) pattern
 - Connections to streams, React

Design Goals

- What are we looking for in design?

Design Goals

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

Design & Concurrency

- So far, mostly **low-level** concurrency idioms
 - What design challenges do we face?
- Two case-studies
 - Code examples off-slides

A simple function

...in sync world

```
function copyFileSync(source: string, dest: string) {
    // Stat dest.
    try {
        fs.statSync(dest);
    } catch {
        console.log("Destination already exists")
        return;
    }

    // Open source.
    let fd;
    try {
        fd = fs.openSync(source, 'r');
    } catch {
        console.log("Destination already exists")
        return;
    }

    // Read source.
    let buff = Buffer.alloc(1000)
    try {
        fs.readSync(fd, buff, 0, 0, 1000);
    } catch (_) {
        console.log("Could not read source file")
        return;
    }

    // Write to dest.
    try {
        fs.writeFileSync(dest, buff)
    } catch (_) {
        console.log("Failed to write to dest")
    }
}
```

Design Goals

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

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?

```
function copyFileSync(source: string, dest: string) {  
    // Stat dest.  
    try {  
        fs.statSync(dest);  
    } catch {  
        console.log("Destination already exists")  
        return;  
    }  
  
    // Open source.  
    let fd;  
    try {  
        fd = fs.openSync(source, 'r');  
    } catch {  
        console.log("Destination already exists")  
        return;  
    }  
  
    // Read source.  
    let buff = Buffer.alloc(1000)  
    try {  
        fs.readSync(fd, buff, 0, 0, 1000);  
    } catch (_) {  
        console.log("Could not read source file")  
        return;  
    }  
  
    // Write to dest.  
    try {  
        fs.writeFileSync(dest, buff)  
    } catch (_) {  
        console.log("Failed to write to dest")  
    }  
}
```

Event Handling in JS

What if our callbacks need callbacks?

Callback Hell

More than nested functions!

- How to handle conditions (or loops)?
- Managing exceptional behavior in both sync and async code

Design Goals

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

Design Goals

- What are we looking for in design?
 - Reuse
 - Readability
 - Robustness
 - Extensibility
 - **Performance**
 - ...

Promises

- Are immutable
- And available repeatedly to observers
- Compare ‘Future’ in Java
 - ‘CompletableFuture’ is probably closest

Design Goals

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

Design Goals

- What are we looking for in design?
 - Reuse
 - **Readability**
 - Robustness
 - **Extensibility**
 - **Performance**
 - ...

Promises: downsides

- Still heavy syntax
- Hard to trace errors
- Doesn't quite solve complex callbacks
 - E.g., if X, call this, else that

Next Step: Async/Await

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

```
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
  }
}
```

Design Goals

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

Design Goals

- What are we looking for in design?
 - *Reuse*
 - **Readability**
 - **Robustness**
 - **Extensibility**
 - **Performance**
 - ...

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

Promises: Guarantees

- Callbacks are never invoked before the current run of the event loop completes
- Callbacks are always invoked, even if (chronologically) added after asynchronous operation completes
- Multiple callbacks are called in order

Design for Concurrency

Let's squint at a few similar developments

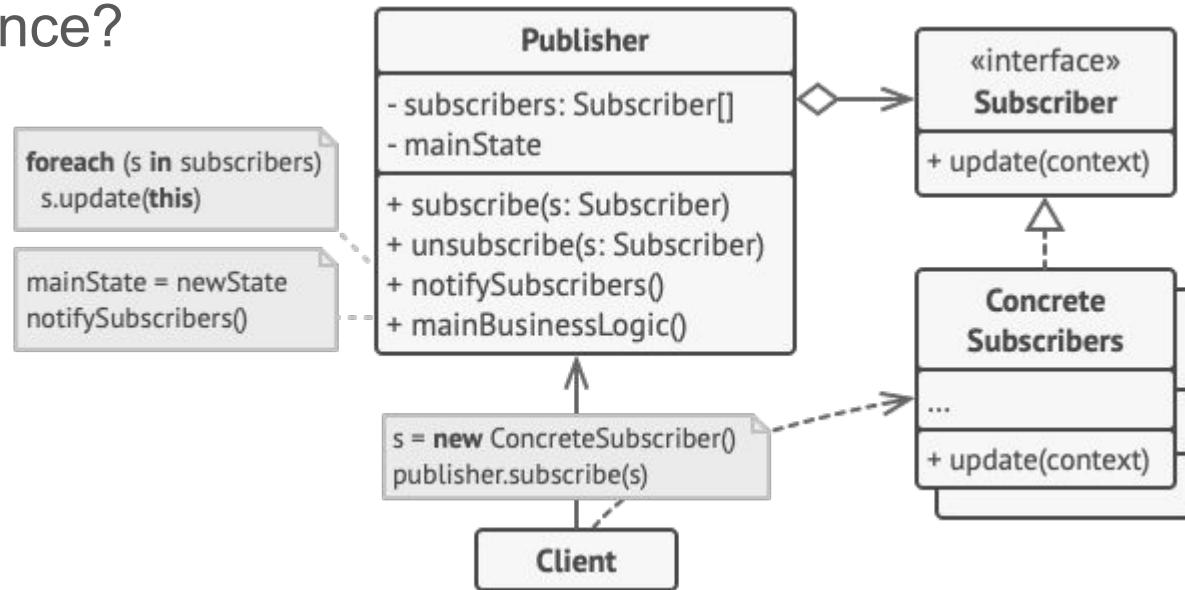
Generator Pattern

- 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?

Observer Pattern

Recall: let objects observe behavior of others

What is the difference?



<https://refactoring.guru/design-patterns/observer>

Observer vs. Generator

Push vs. Pull

- In Observer, the publisher controls information flow
 - When it pushes, everyone must listen
- In generators, the listener “pulls” elements
 - Generator may only prepare the next element upon/after pull
- Which is better?
 - Generators are in a sense ‘observers’ to their clients.
 - This inversion of control can make flow management easier

Manipulating Data

Problem: processing sequential data without assuming its presence

- Let's assume a list of future ints
- Apply a series of transformations
 - E.g., map/update, filter
- Use the result in some operation
 - E.g., collect, foreach

Manipulating Data

Easy solution: collect it all

- Downsides?

```
public class SyncList {  
  
    private int[] data;  
  
    public SyncList(List<Future<Integer>> ints) throws ExecutionException,  
        InterruptedException {  
        this.data = new int[ints.size()];  
        for (int i = 0; i < ints.size(); i++) {  
            this.data[i] = ints.get(i).get();  
        }  
    }  
  
    public void map(Function<Integer, Integer> mapper) {  
        for (int i = 0; i < this.data.length; i++) {  
            this.data[i] = mapper.apply(this.data[i]);  
        }  
    }  
  
    public void filter(Function<Integer, Boolean> filterer) {  
        int newSize = 0;  
        boolean[] filtered = new boolean[this.data.length];  
        for (int i = 0; i < this.data.length; i++) {  
            filtered[i] = filterer.apply(this.data[i]);  
            if (filtered[i]) newSize++;  
        }  
    }  
}
```

Design Goals

- What are we looking for in design?
 - Reuse
 - **Readability**
 - Robustness
 - **Extensibility**
 - **Performance**
 - ...

Manipulating Data

How about:

```
public class AsyncList implements Closeable {

    private final List<Future<Integer>> values;
    private final ExecutorService executor;

    public AsyncList(List<Future<Integer>> values) {
        this.values = values;
        this.executor = Executors.newSingleThreadExecutor();
    }

    public void map(Function<Integer, Integer> updater) {
        for (int i = 0; i < this.values.size(); i++) {
            Future<Integer> val = this.values.get(i);
            this.values.set(i, this.executor.submit(() -> updater.apply(val.get())));
        }
    }

    public void filter(Function<Integer, Boolean> filter) {
        for (int i = 0; i < this.values.size(); i++) {
            Future<Integer> val = this.values.get(i);
            Future<Boolean> filtered = this.executor.submit(() -> filter.apply(val.get()));
        }
    }
}
```

Design Goals

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

Design Goals

- What are we looking for in design?
 - **Reuse**
 - **Readability**
 - *Robustness*
 - **Extensibility**
 - **Performance**
 - ...

Manipulating Data

How about:

```
abstract class AbstractAsyncLazyList implements AsyncLazyList, Closeable {

    protected final AbstractAsyncLazyList upstream;
    private final ExecutorService executor;

    public AbstractAsyncLazyList(AbstractAsyncLazyList upstream) {
        this.upstream = upstream;
        this.executor = Executors.newSingleThreadExecutor();
    }

    abstract Future<Integer> nextValue();

    public AsyncLazyList map(Function<Integer, Integer> mapper) {
        return new MapLazyList( upstream: this, mapper);
    }

    public AsyncLazyList filter(Function<Integer, Boolean> filter) {
        return new FilterLazyList( upstream: this, filter);
    }

    public List<Integer> collect() {
        List<Integer> result = new ArrayList<>();
        Future<Integer> value;
        while ((value = this.nextValue()) != null) {
            result.add(value.get());
        }
        return result;
    }
}
```

Design Goals

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

Design Goals

- What are we looking for in design?
 - **Reuse**
 - **Readability**
 - **Robustness**
 - **Extensibility**
 - **Performance**
 - ...

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`

```
public interface Iterable<E> {
    public Iterator<E> iterator();
}

public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

- In JavaScript (ES6)

```
let arguments = ...
for (const s of arguments) {
    console.log(s)
}
```

- Works for every implementation with a “magic” function `[Symbol.iterator]` providing an iterator

```
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;
}
```

The Iterator Idea

Iterate over elements in arbitrary data structures (lists, sets, trees)
without having to know internals

Typical interface:

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
}
```

(in Java also remove)

Using an iterator

Can be used explicitly

```
List<String> arguments = ...;  
for (Iterator<String> it = arguments.iterator(); it.hasNext(); ) {  
    String s = it.next();  
    System.out.println(s);  
}
```

Often used with magic syntax:

```
for (String s : arguments)  
for (const s of arguments)
```

Java: Getting an Iterator

```
public interface Collection<E> extends Iterable<E> {  
    boolean add(E e);  
    boolean addAll(Collection<? extends E> c);  
    boolean remove(Object e);  
    boolean removeAll(Collection<?> c);  
    boolean retainAll(Collection<?> c);  
    boolean contains(Object e);  
    boolean containsAll(Collection<?> c);  
    void clear();  
    int size();  
    boolean isEmpty();  
    Iterator<E> iterator();  
    Object[] toArray();  
    <T> T[] toArray(T[] a);  
    ...  
}
```

*Defines an interface for creating an Iterator,
but allows Collection implementation to decide
which Iterator to create.*

Iterators for everything

```
public class Pair<E> {  
    private final E first, second;  
    public Pair(E f, E s) { first = f; second = s; }  
  
}
```

```
Pair<String> pair = new Pair<String>("foo", "bar");  
for (String s : pair) { ... }
```

An Iterator implementation for Pairs

```
public class Pair<E> implements Iterable<E> {
    private final E first, second;
    public Pair(E f, E s) { first = f; second = s; }
    public Iterator<E> iterator() {
        return new PairIterator();
    }
    private class PairIterator implements Iterator<E> {
        private boolean seenFirst = false, seenSecond = false;
        public boolean hasNext() { return !seenSecond; }
        public E next() {
            if (!seenFirst) { seenFirst = true; return first; }
            if (!seenSecond) { seenSecond = true; return second; }
            throw new NoSuchElementException();
        }
        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
}
Pair<String> pair = new Pair<String>("foo", "bar");
for (String s : pair) { ... }
```

Iterator design pattern

- Problem: Clients need uniform strategy to access all elements in a container, independent of the container type
 - Order is unspecified, but access every element once
- Solution: A strategy pattern for iteration
- Consequences:
 - Hides internal implementation of underlying container
 - Easy to change container type
 - Facilitates communication between parts of the program

Streams

- Stream is like an Iterator
 - A sequence of objects
 - *Not* interested in accessing specific addresses
- Typically provide operations
 - To translate stream: map, flatMap, filter
 - Operations on all elements (fold, sum) with higher-order functions
 - Often provide efficient/parallel implementations (subtype polymorphism)
- Built-in in Java since Java 8; basics in Node libraries in JS

```
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<Person>.filter({age: 23}).flatMap("children").map("firstName")
    .distinct().filter(/a.*/i).join(", ");
```

A Glimpse at Reactive Programming

(not to be confused with ReactJS)

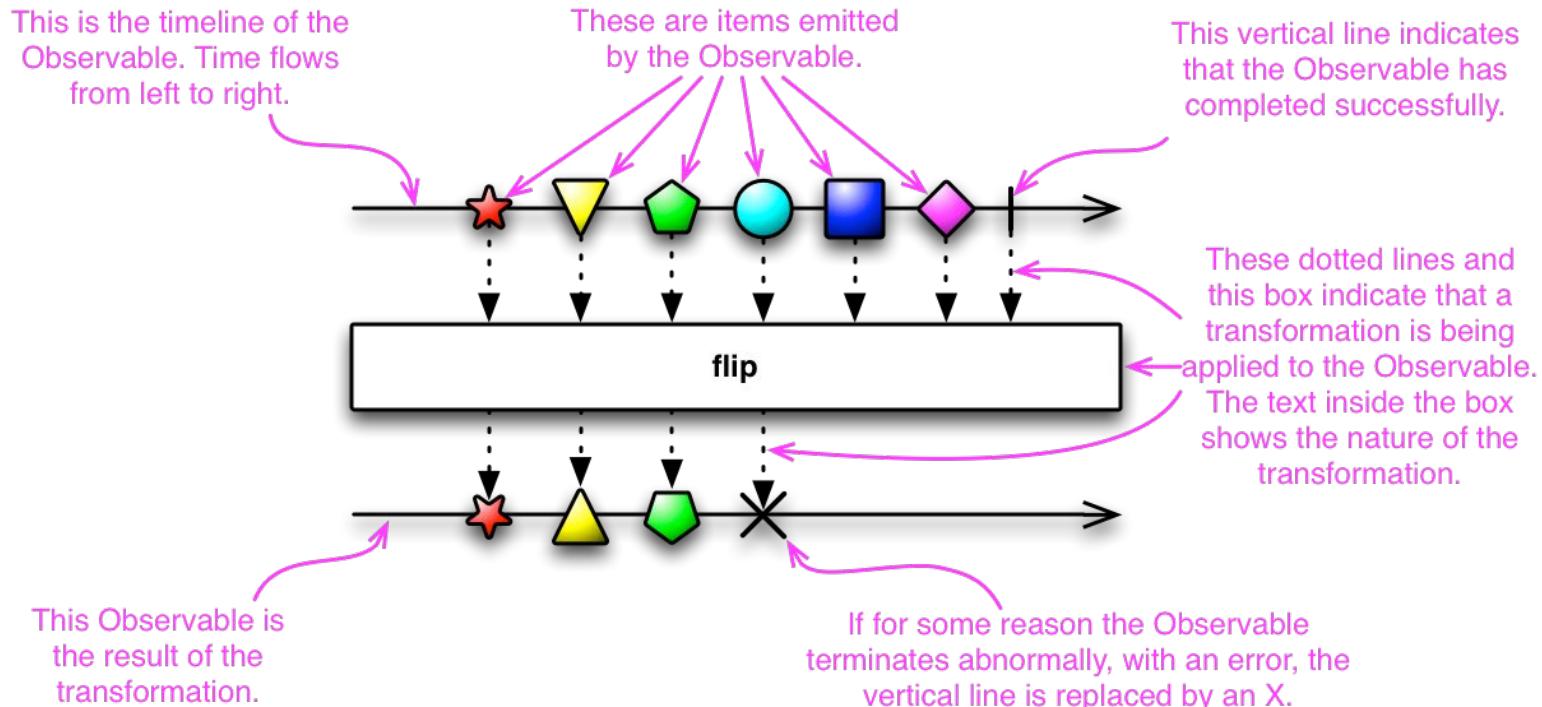
Observable

Observables are lazy Push collections of multiple values. They fill the missing spot in the following table:

SINGLE	MULTIPLE	
Pull	Function	Iterator
Push	Promise	Observable

<https://rxjs.dev/guide/observable>

A Glimpse at Reactive Programming



<http://reactivex.io/documentation/observable.html>

A Glimpse at Reactive Programming

(not to be confused with ReactJS)

- Rx Observables
 - Similar to “standard” observers
 - “An Observable is just the Observer pattern with a jetpack”*
 - Combined with a rich set of *operators*
 - Compare the stream library, times a lot
 - And *flow-control* in the form of back-pressure
 - Makes for a unified API for polymorphic asynchronous events

*<https://x-team.com/blog/rxjs-observables/>

Summary

- Concurrency brings unique design problems
 - And patterns
 - Promises are a key one
 - Worth understanding relations to (async) generators, streams

Self-Assess In-Class Participation

<https://bit.ly/214selfpart>



Designing for Concurrency

- Previously: synchronization of methods, variable read/writes
 - Is that enough?

Designing for Concurrency

- Previously: synchronization of methods, variable read/writes
 - Is that enough?

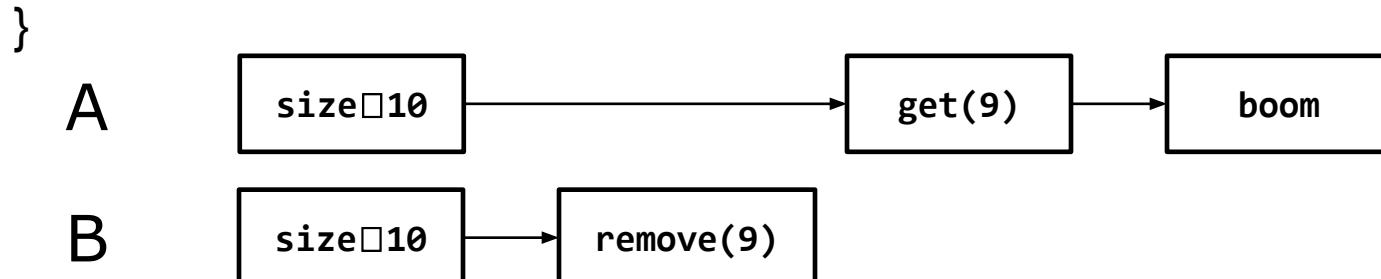
```
public static Object getLast(Vector list) {  
    int lastIndex = list.size() - 1;  
    return list.get(lastIndex);  
}
```

```
public static void deleteLast(Vector list) {  
    int lastIndex = list.size() - 1;  
    list.remove(lastIndex);  
}
```

Object-level concurrency

```
public static Object getLast(Vector list) {  
    int lastIndex = list.size() - 1;  
    return list.get(lastIndex);  
}
```

```
public static void deleteLast(Vector list) {  
    int lastIndex = list.size() - 1;  
    list.remove(lastIndex);  
}
```



Object-level concurrency

Client-side synchronization

```
public static Object getLast(Vector list) {  
    synchronized (list) {  
        int lastIndex = list.size() - 1;  
        return list.get(lastIndex);  
    }  
}
```

```
public static void deleteLast(Vector list) {  
    synchronized (list) {  
        int lastIndex = list.size() - 1;  
        list.remove(lastIndex);  
    }  
}
```

Object-level concurrency

What is the risk here?

```
for (int i = 0; i < vector.size(); i++)  
    doSomething(vector.get(i));
```

Object-level concurrency

What is the risk here?

```
for (int i = 0; i < vector.size(); i++)  
    doSomething(vector.get(i));
```

```
synchronized (vector) {  
    for (int i = 0; i < vector.size(); i++)  
        doSomething(vector.get(i));  
}
```

Object-level concurrency

A common mistake:

```
public Object setup() {  
    if (obj == null) {  
        synchronized (this) {  
            obj = this.initializeObject()  
        }  
    }  
}
```

Principles of Software Construction: Objects, Design, and Concurrency

Events Everywhere

Christian Kästner Vincent Hellendoorn



Outline

- Revisiting Immutability
- Model-View-Controller
- Event-Based Programming, Reactive Programming
- ReactJS UI

Revisiting Immutability

Reading Quiz: Immutability

<https://bit.ly/214q1026>



Recall: Why Immutability?

Recall: 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

Immutable?

```
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  
    }  
}
```

Immutable?

Inner mutable state
(List in Java)

Create copy of
mutable object
(new ArrayList(old)
in Java)

Return new
immutable object

```
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  
    }  
}
```

Aliasing is what makes Mutable State risky

Many variables may point to same object

Any reference to the object can modify the object, effect seen by all other users

x, y, and z all point to
the same mutable
array

```
const x = [ 1, 2, 3 ]
const y = x
function foo(z: number[]): void { /*...*/ }
foo(y)
```

Immutable?

Inner mutable state
(List in Java)

Create copy of
mutable object
(new ArrayList(old)
in Java)

Return new
immutable object

Leak mutable state
Accept mutable state

```
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  
    }  
}
```

Fixed

```
class Stack {  
    readonly #inner: any[]  
    constructor (inner: any[]) {  
        this.#inner=inner.slice()  
    }  
    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.slice()  
        // Java: return new ArrayList(inner)  
    }  
}
```

Recall: 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**

Writing Immutable Data Structures

Any “set” operation returns a new copy of an object
(can point to old object to save memory, e.g. linked lists)

Final fields of immutable objects are save (e.g., strings, numbers)

Fields of mutable objects must be protected
(encapsulation, making copies)

Careful with mutable constructor/method arguments (make copies)

Easy to make mistakes when mixing mutable and immutable data structures, only academic tools for checking

Trend toward immutable data structures

Immutable data structures common in functional programming

Many recent languages and libraries embrace immutability

Scala, Rust, stream, React, Java Records

Simplifies building concurrent and distributed systems

Requires some practice when used to imperative programming with mutable state, but will become natural

Circular references & Caching

Immutable data structures often from a directed acyclic graph

Cycles challenging

Cycles often useful for performance (caching)

```
class TreeNode {  
    readonly #parent: TreeNode  
    readonly #children: TreeNode[]  
    constructor(parent: TreeNode,  
               children: TreeNode[]) {  
        this.#parent = parent  
        this.#children = children  
    }  
    addChild(child: TreeNode) {  
        const newChildren = this.#children.slice()  
        //const newChild = child.setParent(this) ??  
        newChildren.push(child)  
        const newNode = new TreeNode(this.#parent,  
                                    newChildren)  
        //child.setParent(newNode) ??  
        return newNode  
    }  
}
```

Design Discussion

Design for Understandability / Maintainability

- Immutable objects are easy to reason about, they won't change
- Mutable objects have more complicated contracts, function and client both can modify state
- Do not need to think about corner cases of concurrent modification

Design for Reuse

- Easy to reuse even in concurrent settings

Java 16 Records

Records are (shallowly) immutable

No setters

But also no defensive copying of mutable fields

Reactive Programming

Reactive Programming

Programming strategy or patterns, where programs react to data

Embraces concurrency, focuses on data flows

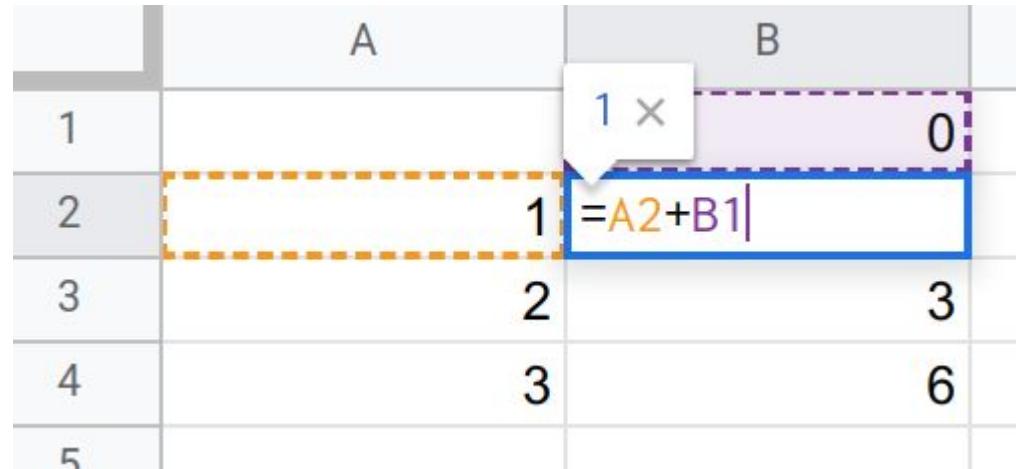
Takes event-based programming to an extreme

Decouples programs around data

Useful analogy: Spreadsheets

Cells contain data or formulas

Formula cells are computed automatically whenever input data changes



A screenshot of a spreadsheet application illustrating how formulas are calculated. The spreadsheet has two columns labeled 'A' and 'B'. Column A contains numerical values from 1 to 5. Column B contains numerical values from 0 to 6. Cell B2 is highlighted with a blue border and contains the formula $=A2+B1$. A small white callout box with a blue border and a purple arrow points to the cell B2, containing the text "1 ×". The formula bar at the top of the cell also shows the formula $=A2+B1$.

	A	B
1		0
2	1	1
3	2	2
4	3	3
5	4	6

Implementing Spreadsheet-Like Computations?

Implementing Spreadsheet-Like Computations?

```
x = 3  
y = 5  
z = x + y  
print(z) // prints 8  
  
x = 5  
print(z) // expect 10, prints 8
```

in imperative computations,
no update when inputs change

Implementing Spreadsheet-Like Computations?

```
x = 3  
y = 5  
z = () => x + y  
print(z()) // prints 8  
x = 5  
print(z()) // prints 10
```

computation performed on demand (pull)
caching possible

Does not easily work in Java, since Java requires variables in closure to be final. Need object with mutable internal state

Implementing Spreadsheet-Like Computations?

```
x = new Cell(3)
y = new Cell(5)

z = new DerivedCell(x, y, (a,b)=>a+b)

print(z.get()) // prints 8

x.set(5)

print(z.get()) // prints 10
```

Cell implements observer pattern,
informs observers of changes (push)

DerivedCell listens to changes from Cell,
updates internal state on changes,
informs own observers of changes

Complications

Single change in cell can trigger many computations (push)

Possibly put in queue, compute asynchronously

Perform some computations lazily when needed

Cyclic dependencies can result in infinite loops

Detect, special ways to handle

Observers can hinder garbage collection

	A	B
1		0
2	1	1
3	2	3
4	3	6
5	#REF!	=B4+3+A5
6		

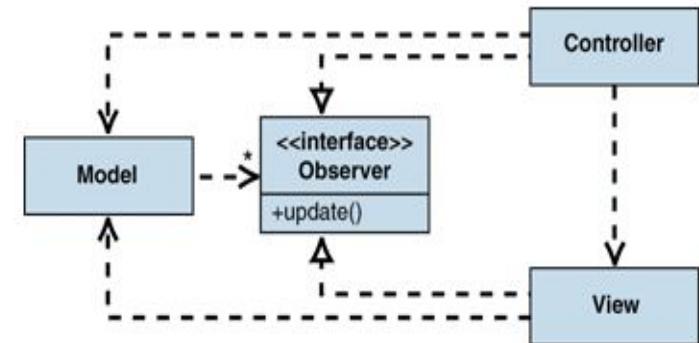
Reactive Programming and GUIs

Store state in observable cells, possibly derived

Have GUI update automatically on state changes

Have buttons perform state changes on cells

Mirrors active model-view-controller pattern, discussed later
(model is observable cell)



From Pull to Push

Instead of clients to look for state (pull)

observers react to state changes with actions (push)

Commonly observables indicate that something has changed, triggering observers to get updated state (push-pull)

Beyond Spreadsheet Cells

SINGLE	MULTIPLE
Pull	Function Iterator
Push	Promise Observable

<https://rxjs.dev/guide/observable>

Reactive Programming Libraries

RxJava, RxJS, many others

Provide Stream-like interfaces for event handling, with many convenience functions (similar to promises)

Observables typically allow pushing multiple values in sequence

Cells can be implemented by considering only the latest value of observables

Previous Example with RxJava

```
PublishSubject<Integer> x = PublishSubject.create();
PublishSubject<Integer> y = PublishSubject.create();
Observable<Integer> z = Observable.combineLatest(x, y,
(a,b)->a+b);
z.subscribe(System.out::println);
x.onNext(3);
y.onNext(5);
x.onNext(5);
```

Chaining Computations along Data

```
awk '{print $7}' < /var/log/nginx/access.log |  
    sort |  
    uniq -c |  
    sort -r -n |  
    head -n 5 > out
```

Multiple programs executed in sequence each read lines and produce lines;
can start reading lines before previous program is finished

Streams / Reactive Programming / Events

Instead of calling methods in sequence,
set up pipelines for data processing

Let data control the execution

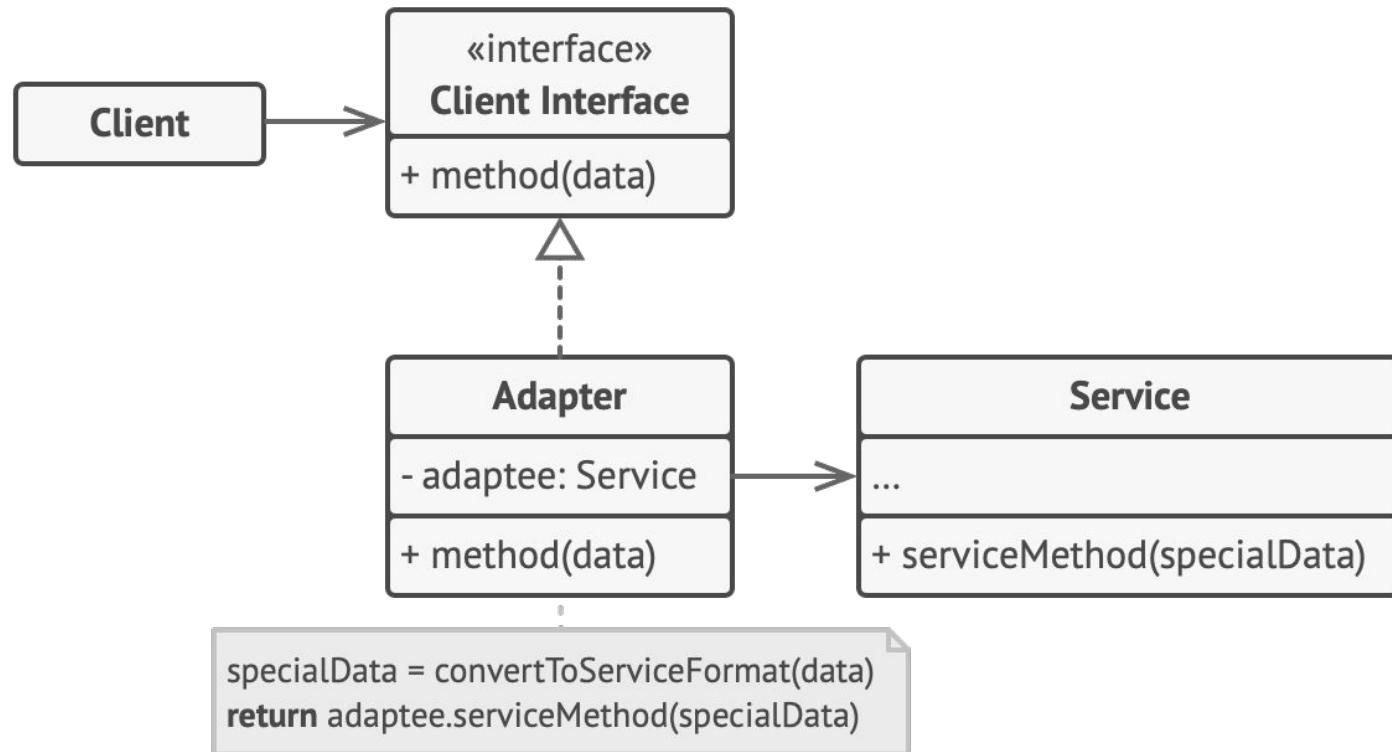
```
var lines = IOHelper.readLinesFromFile(file);
var linesObs = Observable.fromIterable(lines);
linesObs.
    map(Parser::getURLColumn).
    groupBy(...).
    sorted(comparator).
    subscribe(IOHelper.writeToFile(outFile));
```

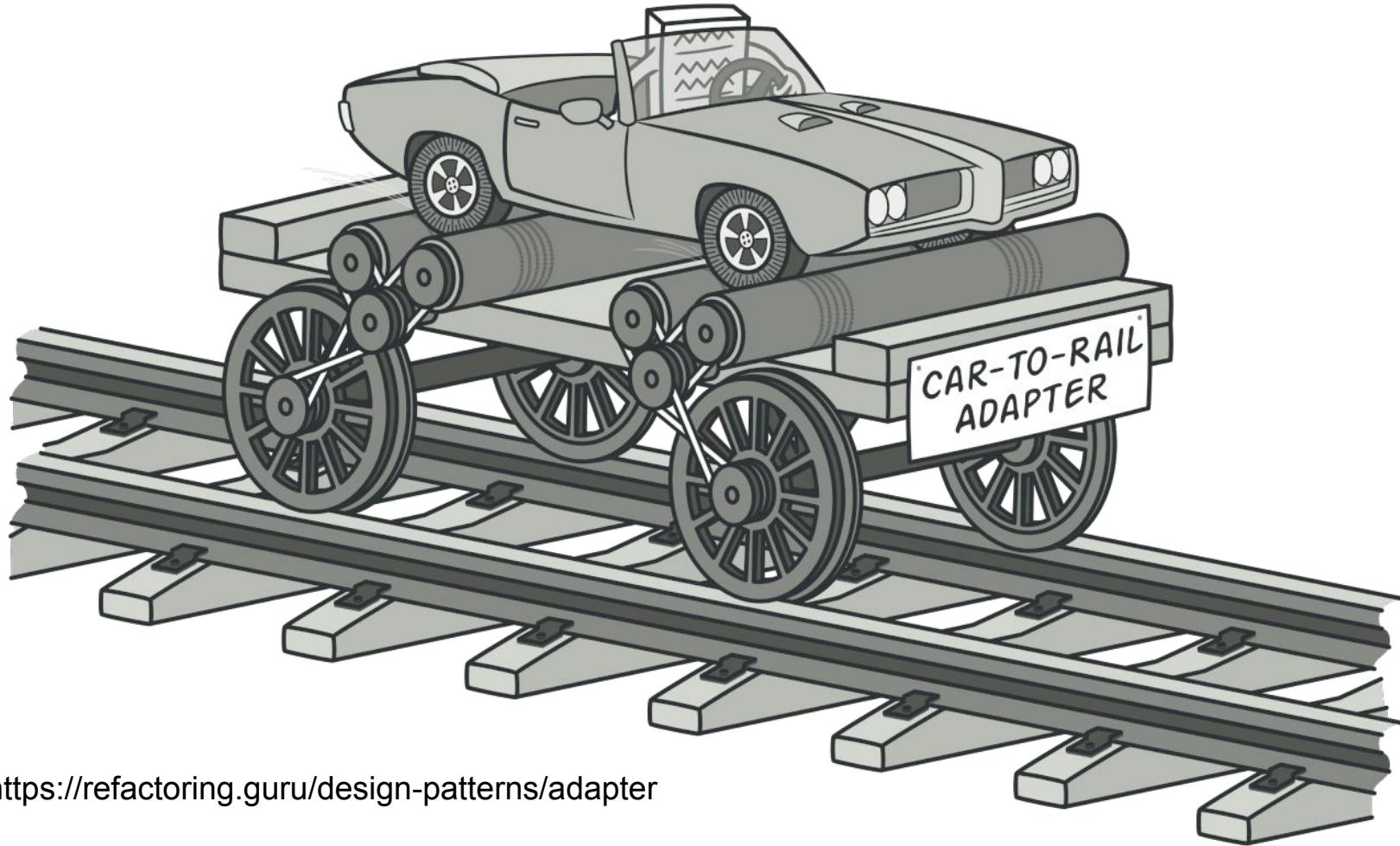
Many more Features in Reactive Programming Libraries

Backpressure (see last lecture)

Aside: The Adapter Pattern

The Adapter Design Pattern



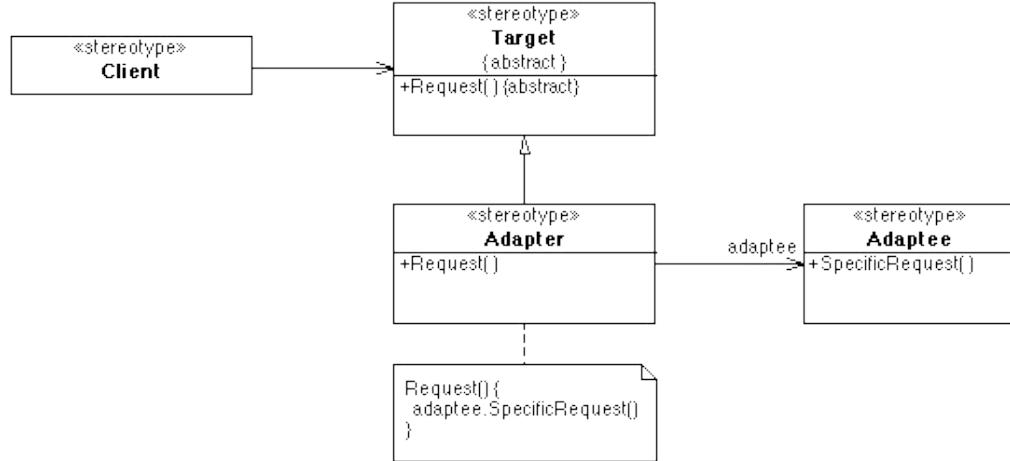


<https://refactoring.guru/design-patterns/adapter>

The Adapter Design Pattern

Applicability

- You want to use an existing class, and its interface does not match the one you need
- You want to create a reusable class that cooperates with unrelated classes that don't necessarily have compatible interfaces
- You need to use several subclasses, but it's impractical to adapt their interface by subclassing each one



Consequences

- Exposes the functionality of an object in another form
- Unifies the interfaces of multiple incompatible adaptee objects
- Lets a single adapter work with multiple adaptees in a hierarchy
- -> **Low coupling, high cohesion**

Adapters for Collections/Streams/Observables

```
var lines = IOHelper.readLinesFromFile(file);
var linesObs = Observable.fromIterable(lines);
linesObs.
    map(Parser::getURLColumn).
    groupBy(...).
    sorted(comparator).
    subscribe(IOHelper.writeToFile(outFile));
```

Any others?

Façade/Controller vs. Adapter

- Motivation
 - Façade: simplify the interface
 - Adapter: match an existing interface
- Adapter: interface is given
 - Not typically true in Façade
- Adapter: polymorphic
 - Dispatch dynamically to multiple implementations
 - Façade: typically choose the implementation statically

Core vs GUI

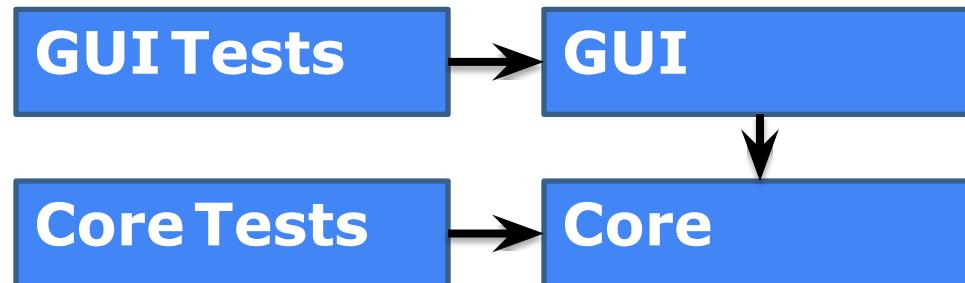
Backend vs Frontend

Recall: Core implementation vs. GUI

- Core implementation: application logic
 - Computing some result, updating data
- GUI
 - Graphical representation of data
 - Source of user interactions
- Design guideline: *avoid coupling the GUI with core application*
 - Multiple UIs with single core implementation
 - Test core without UI

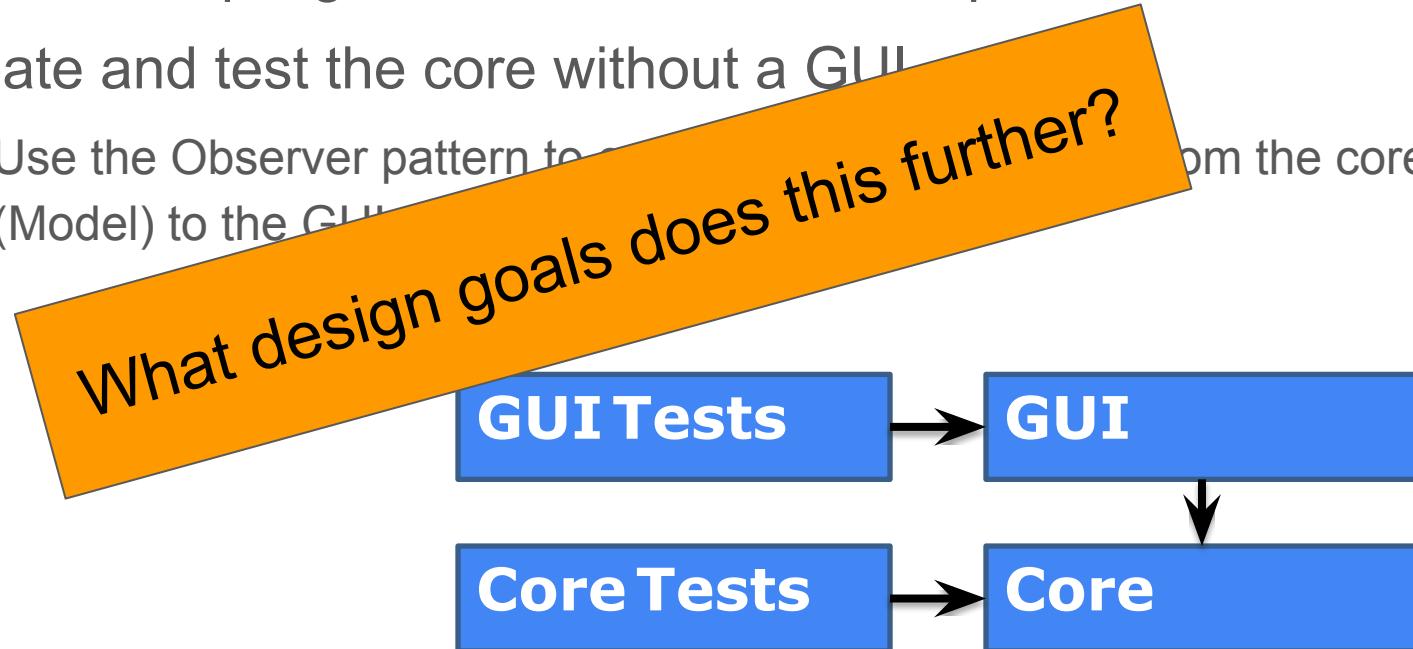
Recall: Separating application core and GUI

- 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)

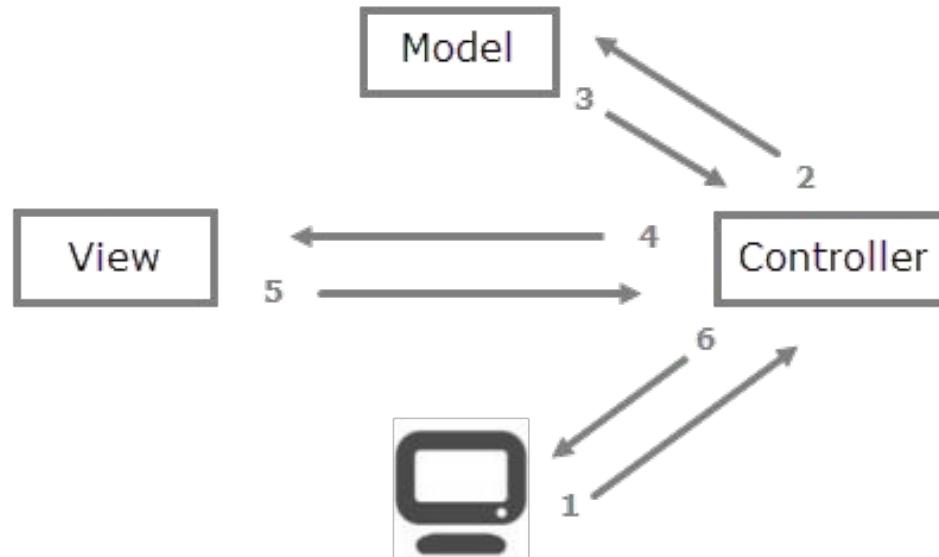


Recall: Separating application core and GUI

- Reduce coupling: do not allow core to depend on UI
- Create and test the core without a GUI
 - Use the Observer pattern to let the GUI (View) depend on the core (Model) to the GUI

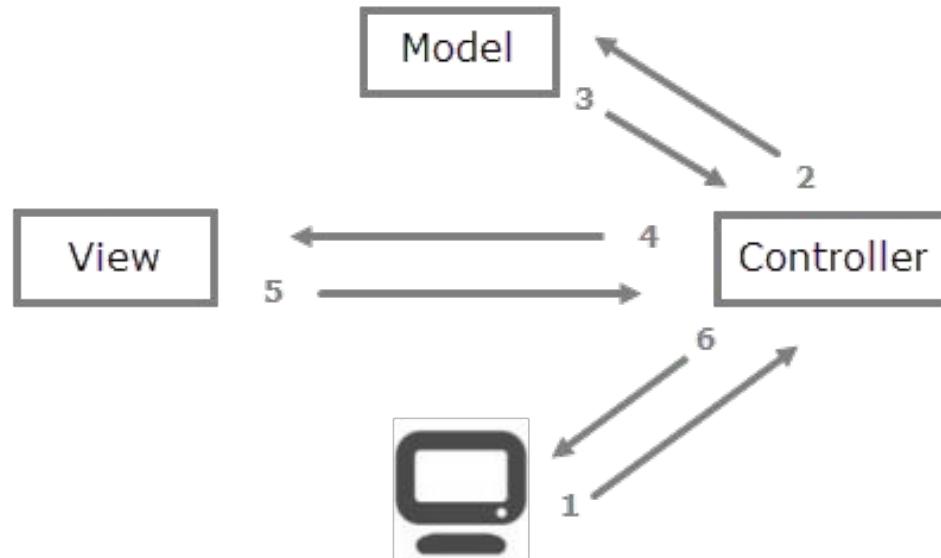


Model View Controller



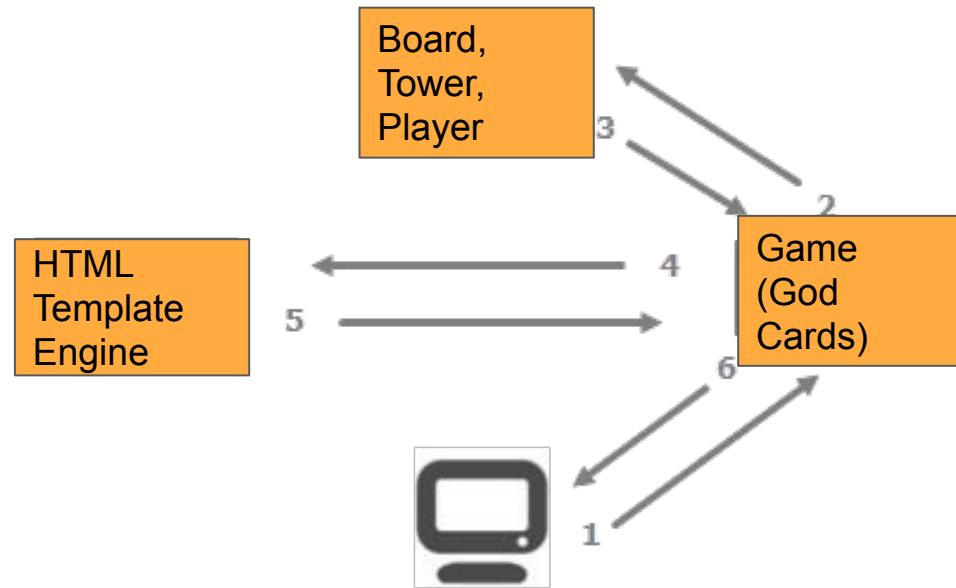
<https://overiq.com/django-1-10/mvc-pattern-and-django/>

Model View Controller in Santorini?



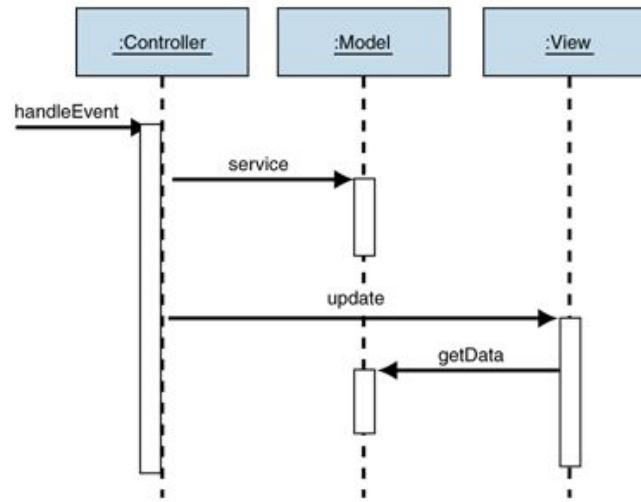
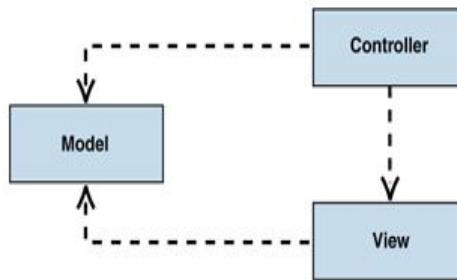
<https://overiq.com/django-1-10/mvc-pattern-and-django/>

Model View Controller in Santorini?

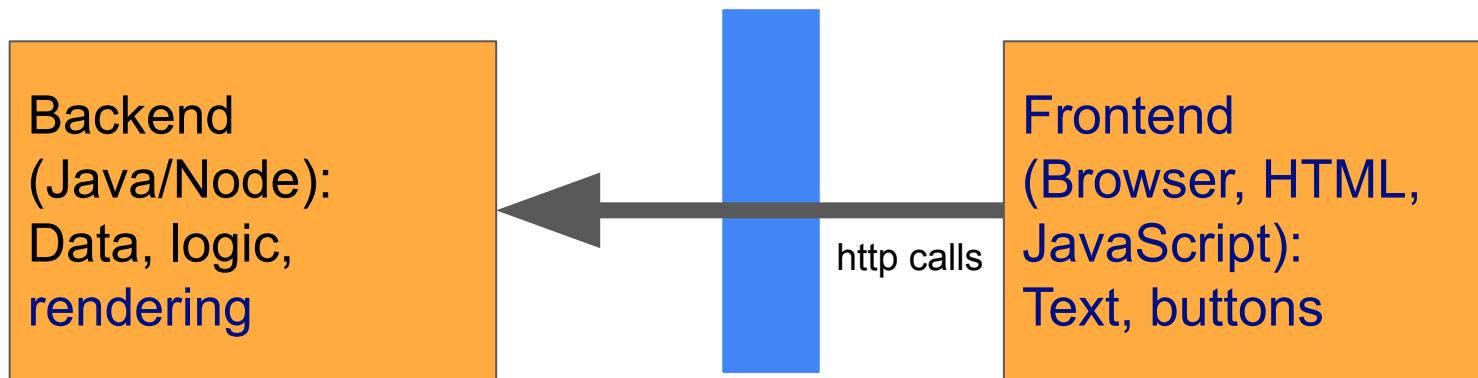


<https://overiq.com/django-1-10/mvc-pattern-and-django/>

Model View Controller Dependencies

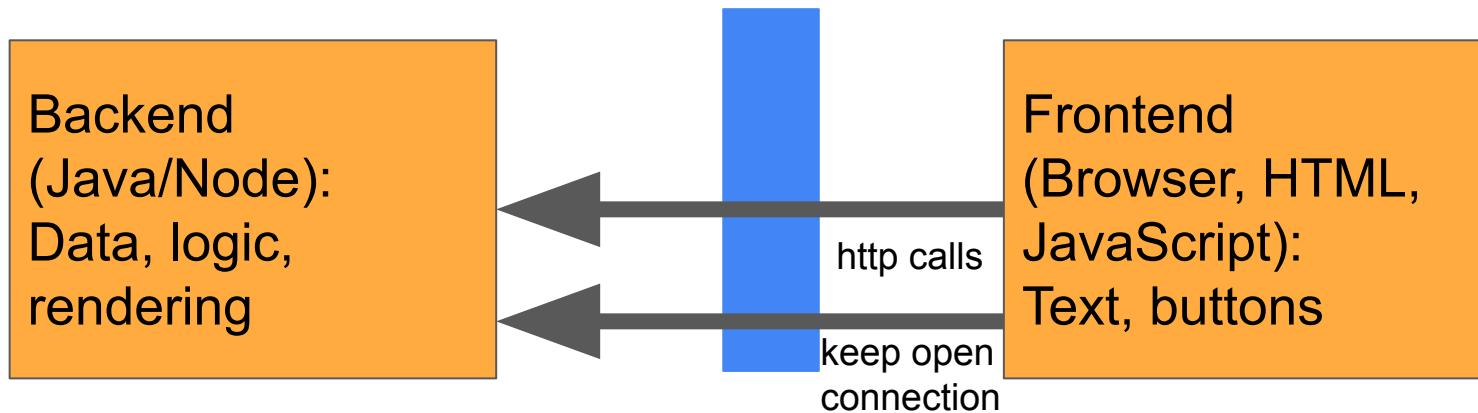


Client-Server Programming forces Frontend-Backend Separation



Browser can call web server, but not the other way around
Browser needs to *pull* for updates
Browser can request entirely no page or just additional content
(ajax, REST api calls, ...)

Client-Server Programming forces Frontend-Backend Separation



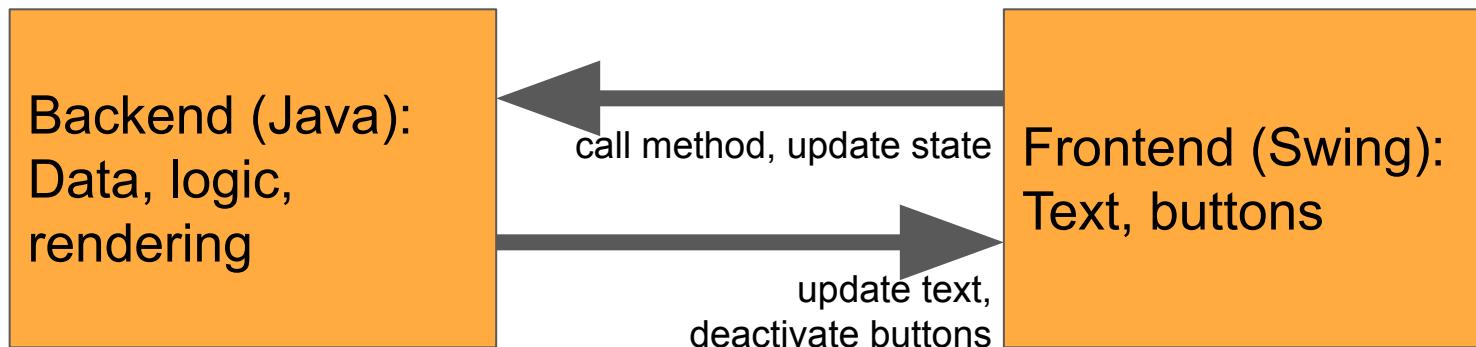
Trick to let backend push information to frontend: Keep http request open, append to page (compare to stream)
Alternative: regular pulling

Core & Gui in same environment

JavaScript frontend and backend together in browser
(e.g. using *browserify*) -- single threaded!

Java Swing GUI running in same VM as core logic -- multi threaded

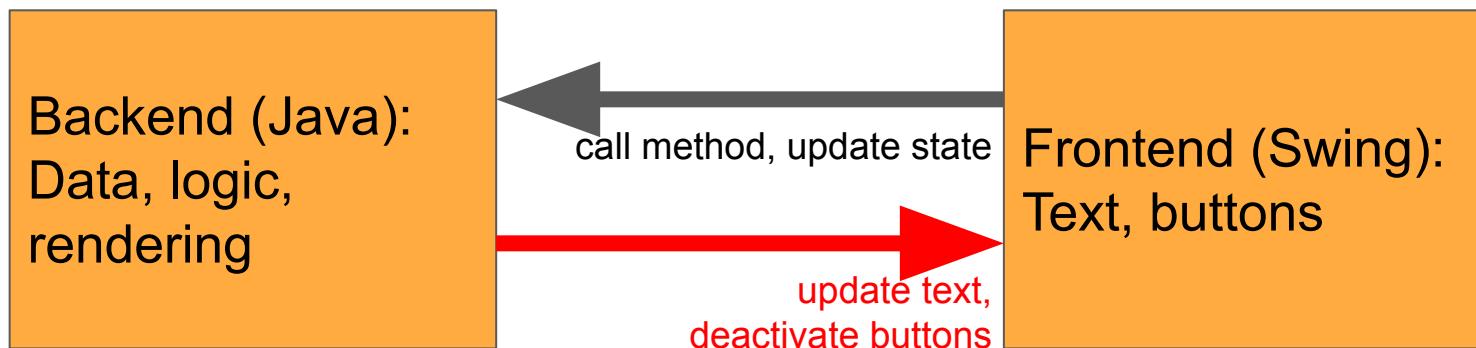
Core logic could directly modify GUI



Avoid Core to Gui coupling

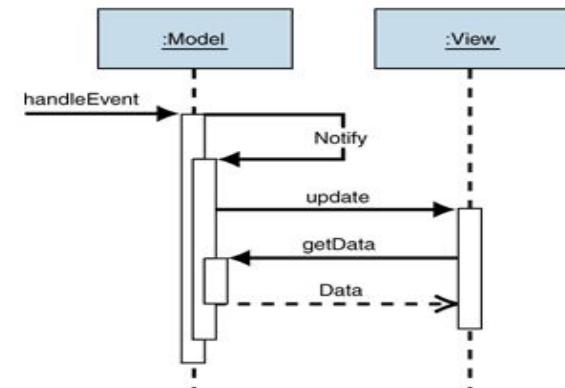
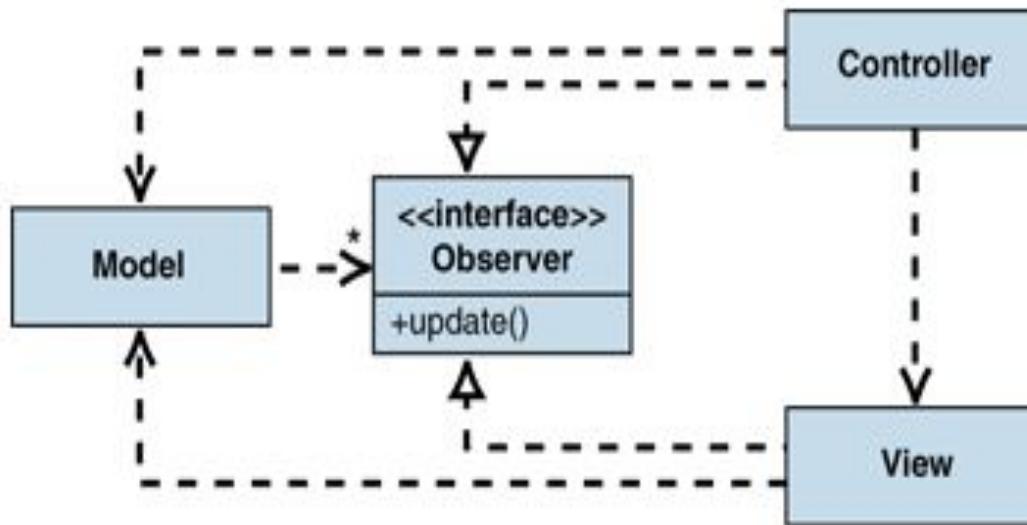
Never call the GUI from the Core

Update GUI after action (pull) or use observer pattern instead to inform GUI of updates (push)



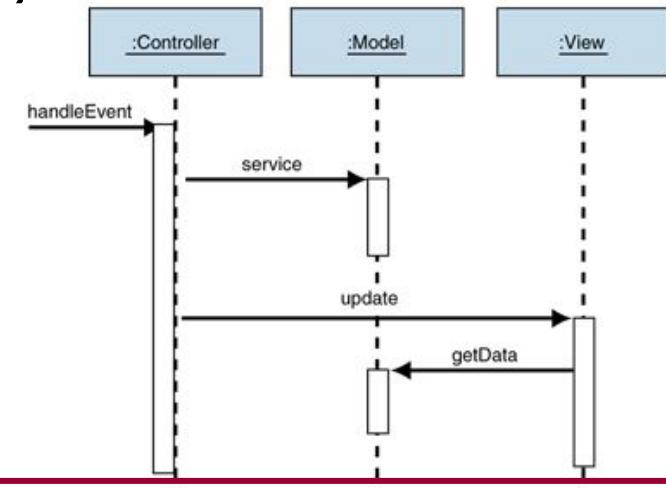
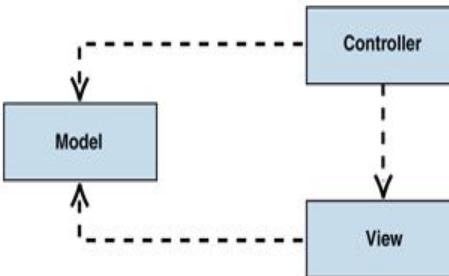
Excursion: Active Model View Controller

(Not commonly used)

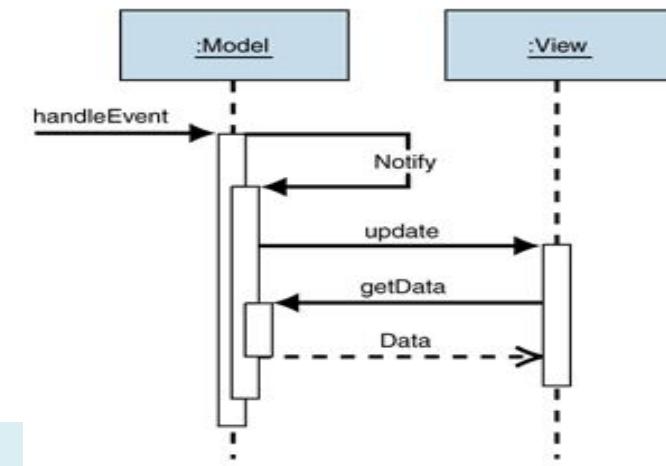
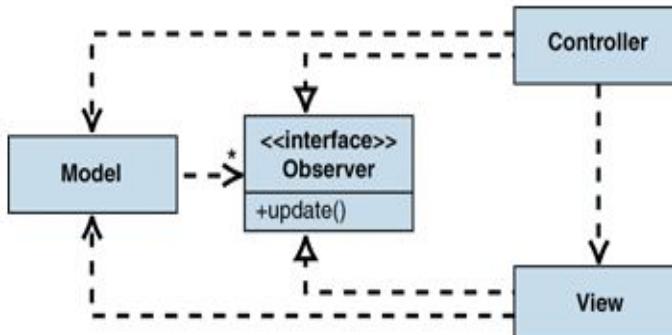


Model-View-Controller (MVC)

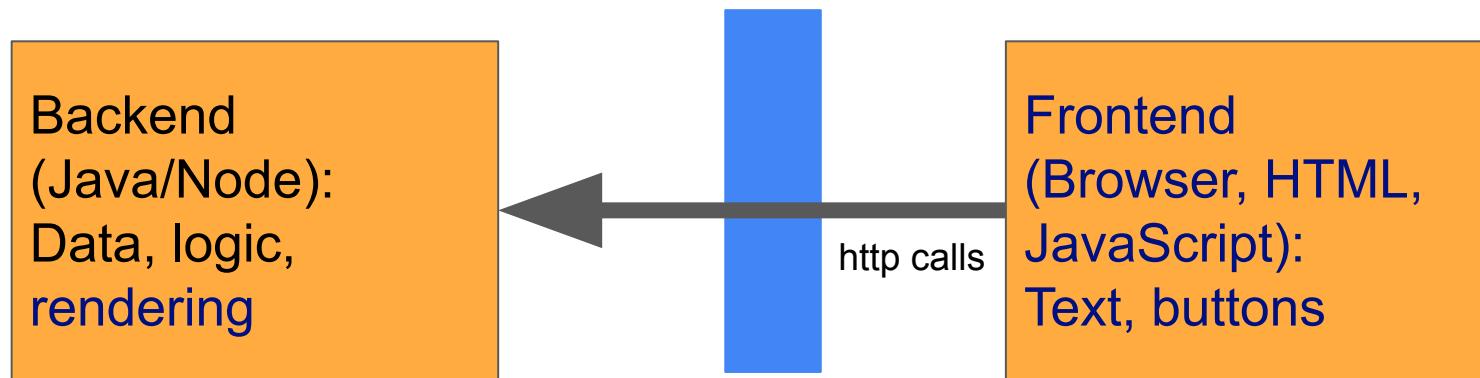
Passive model



Active model



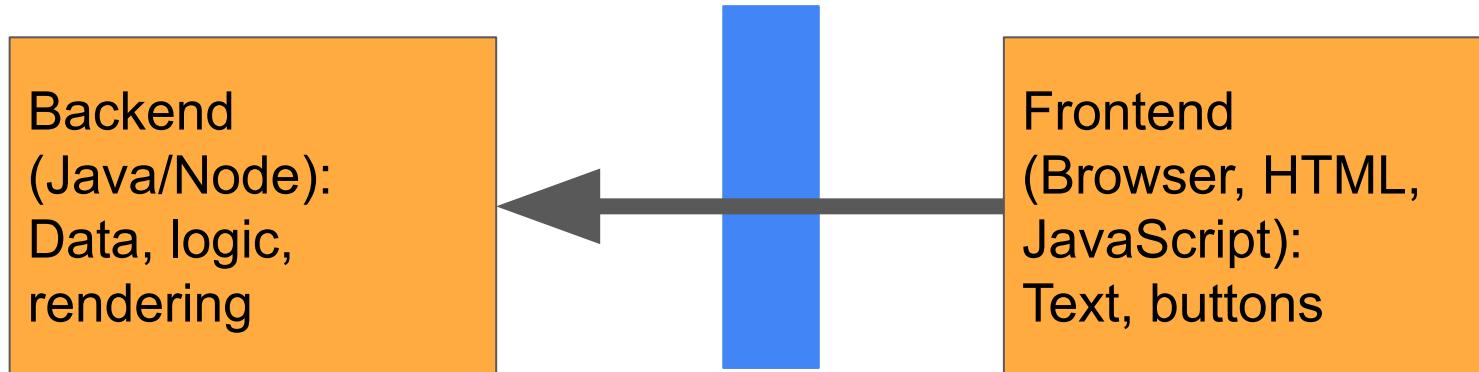
GUI Code in the Backend



Typically there is GUI code in Backend (rendering/view)
Could also send entire program state to frontend (e.g, json) and render there with JavaScript

Where to put GUI Logic?

Example: Deactivate undo button in first round of TicTacToe,
deactivate game buttons after game won

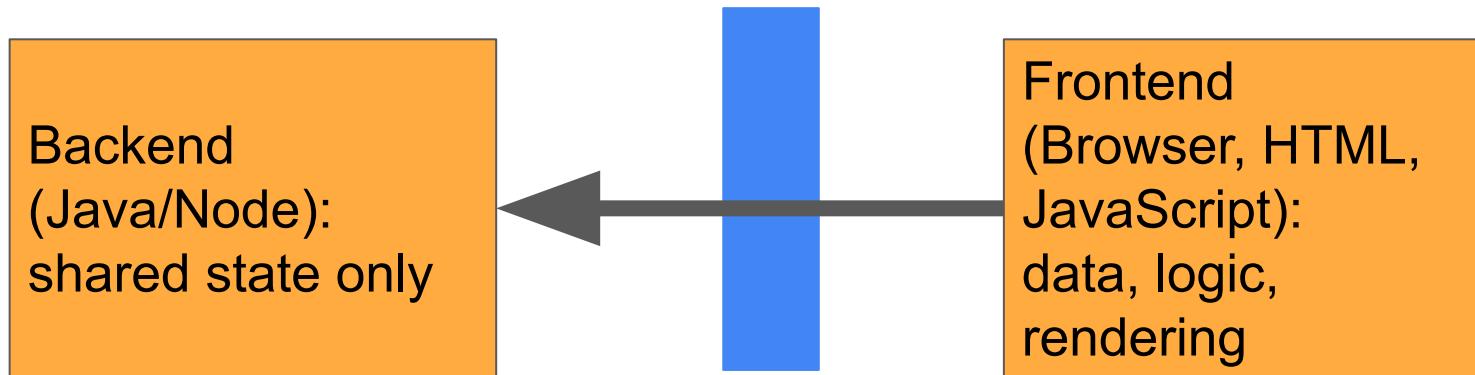


- Option 1: All rendering in backend, update/refresh the entire page after every action -- simpler
- Option 2: Handle some logic in frontend, use backend for checking -- fewer calls, more responsive

Core Logic in Frontend?

Could move core logic largely to client, minimize backend interaction

Can frontend be trusted? Need to replicate core in front and backend?



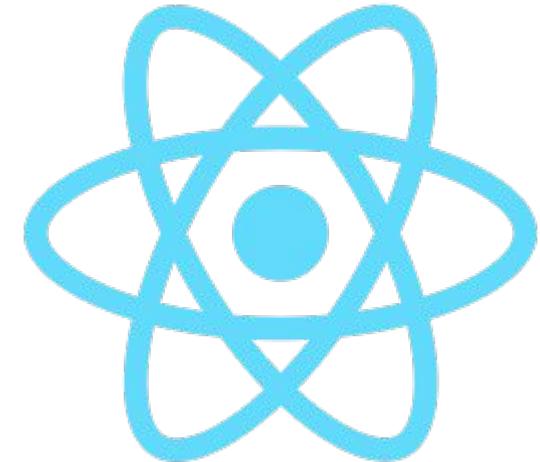
(React and other frameworks make it easy to introduce logic in the frontend; avoid tangling all core logic with GUI)

ReactJS

ReactJS

Popular frontend library by Facebook

Template library and state management



Not reactive programming library, though it adopts some similar ideas

Templates with ReactJS

(Similar ideas to Handlebars in HW4 and Rec7)

Describe rendering of HTML, inputs given as objects

JSX language extension to embed HTML in JS

Try it:

<https://reactjs.org/redirect-to-codepen/introducing-jsx>

```
function formatName(user) {  
    return user.firstName + ' ' +  
        user.lastName;  
}  
  
const user = {  
    firstName: 'Harper',  
    lastName: 'Perez'  
};  
  
const element = (  
    <h1>Hello, {formatName(user)}!</h1>  
);  
  
ReactDOM.render(  
    element,  
    document.getElementById('root')  
);
```

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>

```
function Welcome(props) {  
  return <h1>Hello, {props.name}</h1>;  
}  
  
function App() { return (  
  <div>  
    <Welcome name="Sara" />  
    <Welcome name="Edite" />  
  </div>  
);}  
  
ReactDOM.render(  
  <App />,  
  document.getElementById('root')  
);
```

Templates with State

Class notation instead of function

State is like a cell in reactive programming, *if state changes, page is re-rendered*

Try it:

<https://codepen.io/gaearon/pen/xEmzGg?editors=0010>

```
class Toggle extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {isToggleOn: true};  
    this.handleClick = this.handleClick.bind(this);  
  }  
  handleClick() {  
    this.setState(prevState => ({  
      isToggleOn: !prevState.isToggleOn  
    }));  
  }  
  render() { return (  
    <button onClick={this.handleClick}>  
      {this.state.isToggleOn ? 'ON' : 'OFF'}  
    </button>  
  ); }  
}
```

```
ReactDOM.render(  
  <Toggle />,  
  document.getElementById('root')  
>;
```

ReactJS Templates

Can use arbitrary JavaScript code (Handlebars can only access object properties)

Properties are read-only

State is mutable and *observed* for re-rendering (state updates are asynchronous)

Re-rendering is optimized and asynchronous, will rerender inner components too if their properties change

ReactJS and Core Logic

React makes it easy to add functionality in GUI

This really tangles GUI and logic (violating separation argued for above)

Suggestion: Use React state primarily for UI-related logic (e.g., selecting workers) and keep the core logic in the backend or as a separate library -- be very explicit about what information is shared

Connecting React to backend

Return json from server backend and store as component state

Full example:

<https://www.freecodecamp.org/news/how-to-create-a-react-app-with-a-node-backend-the-complete-guide/>

```
function App() {  
  const [data, setData] =  
    React.useState(null);  
  
  React.useEffect(() => {  
    fetch("/api")  
      .then((res) => res.json())  
      .then((data) =>  
        setData(data.message));  
  }, []);  
  
  return (  
    <div>* using state in data *</div>  
  );  
}
```

React and Homework 5/6

Using React is entirely optional

We use Handlebars by default (HW4 and Rec7)

Many other template engines and frontend frameworks exists (e.g.,
Vue, Angular, ...)

React adds complexity but also easy updates reacting to state changes

Summary

Immutable objects are great! Use them

Reactive programming decouples programs along data
Observer pattern on steroids

New Design Pattern: Adapter

Decompose GUI from Core with Model View Controller Pattern

Brief intro to ReactJS

Principles of Software Construction: Objects, Design, and Concurrency

Libraries and Frameworks

(Design for large-scale reuse)

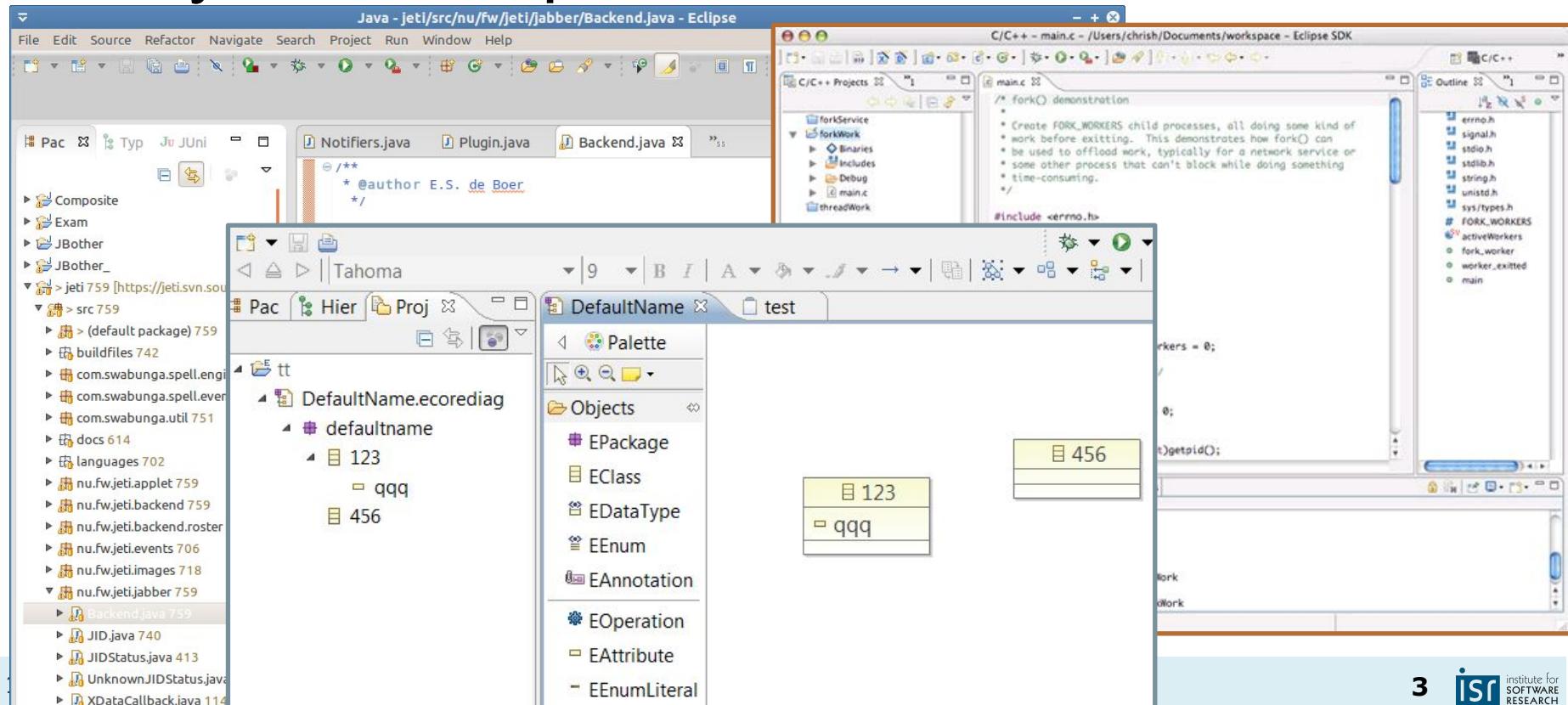
~~Christian Kästner~~ ~~Vincent Hellendoorn~~
Michael Hilton



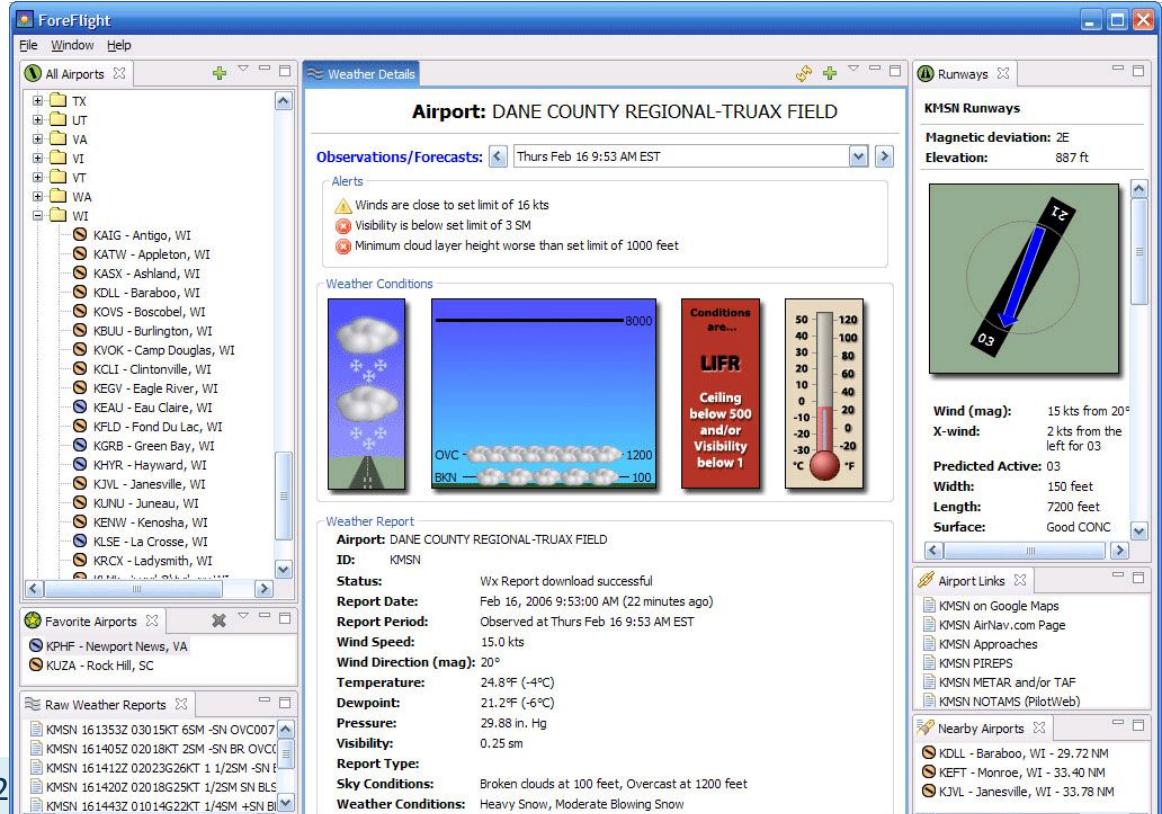
Learning goals for today

- Describe example well-known example frameworks
- Know key terminology related to frameworks
- Know common design patterns in different types of frameworks
- Discuss differences in design trade-offs for libraries vs. frameworks
- Analyze a problem domain to define commonalities and extension points (cold spots and hot spots)
- Analyze trade-offs in the use vs. reuse dilemma
- Know common framework implementation choices

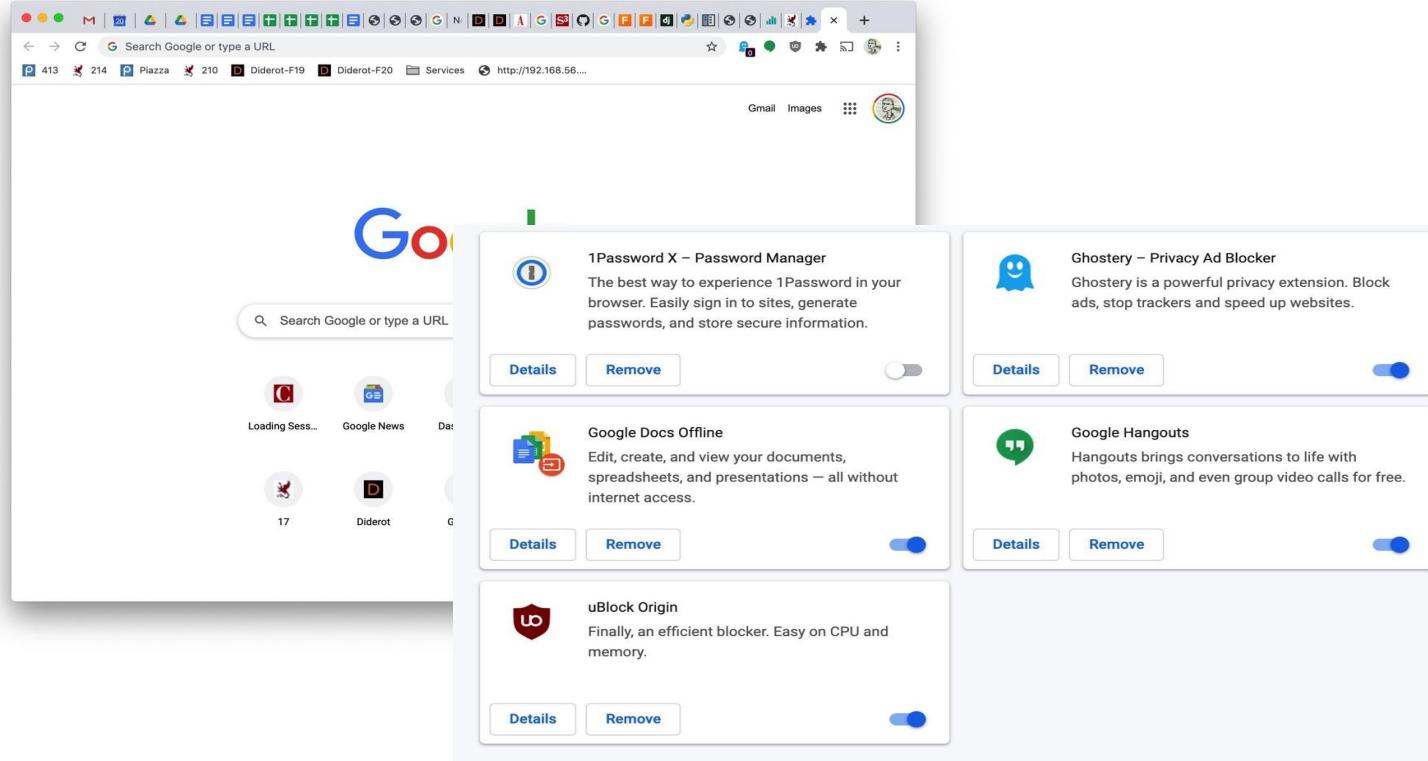
Reuse and variation: Family of development tools



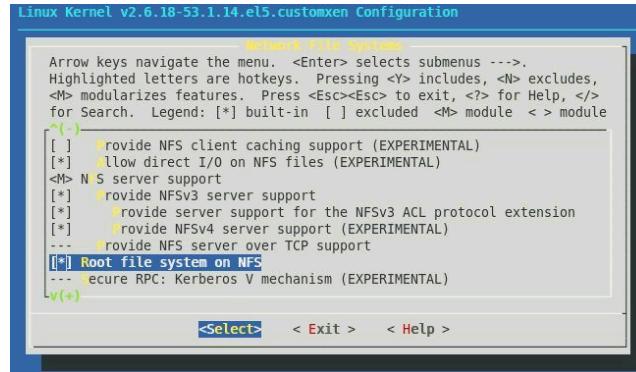
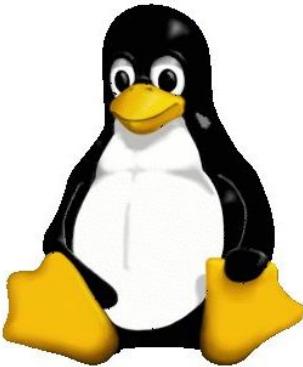
Reuse and variation: Eclipse Rich Client Platform



Reuse and variation: Web browser extensions



Reuse and variation: Flavors of Linux



Reuse and variation: Product lines



Earlier in this course: Class-level reuse

Language mechanisms supporting reuse

- Inheritance
- Subtype polymorphism (dynamic dispatch)
- Parametric polymorphism (generics)

Design principles supporting reuse

- Small interfaces
- Information hiding
- Low coupling
- High cohesion

Design patterns supporting reuse

- Template method, decorator, strategy, composite, adapter, ...

Today: Reuse at scale

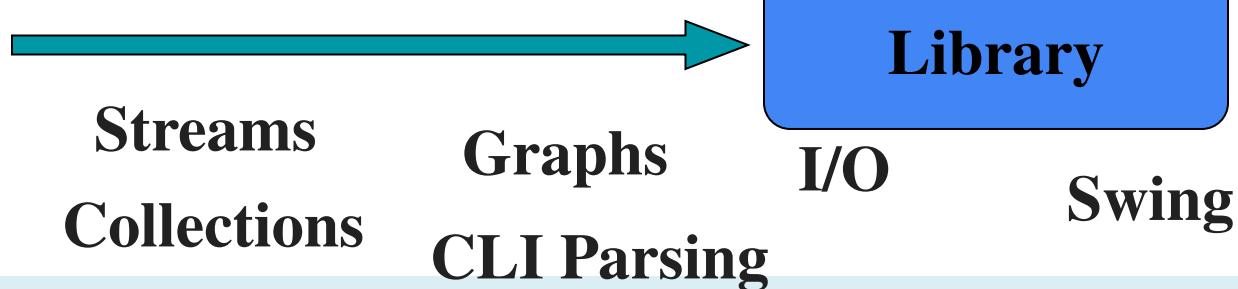
- Examples, terminology
- Whitebox and blackbox frameworks
- Design considerations
- Implementation details
 - Responsibility for running the framework
 - Loading plugins

Terminology: Libraries

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



Math



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.”



```
public MyWidget extends JPanel {  
    public MyWidget(int param) { /* setup  
        internals, without rendering  
    */  
  
        // render component on first view and  
        // resizing  
        protected void  
        paintComponent(Graphics g) {  
            // draw a red box on his  
            componentDimension d = getSize();  
            g.setColor(Color.red);  
            g.drawRect(0, 0, d.getWidth(),  
            d.getHeight());  
        }  
    }  
}
```

your code

Framework

IntelliJ Firefox

Swing

Express NanoHttpd Spring

A calculator example (without a framework)

```
public class Calc extends JFrame {  
    private JTextField textField;  
    public Calc() {  
        JPanel contentPane = new JPanel(new BorderLayout());  
        contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));  
        JButton button = new JButton();  
        button.setText("calculate");  
        contentPane.add(button, BorderLayout.EAST);  
        textField = new JTextField("");  
        textField.setText("10 / 2 + 6");  
        textField.setPreferredSize(new Dimension(200, 20));  
        contentPane.add(textField, BorderLayout.WEST);  
        button.addActionListener(/* calculation code */);  
        this.setContentPane(contentPane);  
        this.pack();  
        this.setLocation(100, 100);  
        this.setTitle("My Great Calculator");  
        ...  
    }  
}
```



A simple example framework

- Consider a family of programs consisting of a button and text field only:



- What source code might be shared?

A calculator example (without a framework)

```
public class Calc extends JFrame {  
    private JTextField textField;  
    public Calc() {  
        JPanel contentPane = new JPanel(new BorderLayout());  
        contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));  
        JButton button = new JButton();  
        button.setText("calculate");  
        contentPane.add(button, BorderLayout.EAST);  
        textField = new JTextField("");  
        textField.setText("10 / 2 + 6");  
        textField.setPreferredSize(new Dimension(200, 20));  
        contentPane.add(textField, BorderLayout.WEST);  
        button.addActionListener(/* calculation code */);  
        this.setContentPane(contentPane);  
        this.pack();  
        this.setLocation(100, 100);  
        this.setTitle("My Great Calculator");
```



A simple example framework

```
public abstract class Application extends JFrame {  
    protected String getApplicationTitle() { return ""; }  
    protected String getButtonText() { return ""; }  
    protected String getInitialText() { return ""; }  
    protected void buttonClicked() {}  
    private JTextField textField;  
    public Application() {  
        JPanel contentPane = new JPanel(new BorderLayout());  
        contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));  
        JButton button = new JButton();  
        button.setText(getButtonText());  
        contentPane.add(button, BorderLayout.EAST);  
        textField = new JTextField("");  
        textField.setText(getInitialText());  
        textField.setPreferredSize(new Dimension(200, 20));  
        contentPane.add(textField, BorderLayout.WEST);  
        button.addActionListener((e) -> { buttonClicked(); });  
        this.setContentPane(contentPane);  
        this.pack();  
    }  
}
```

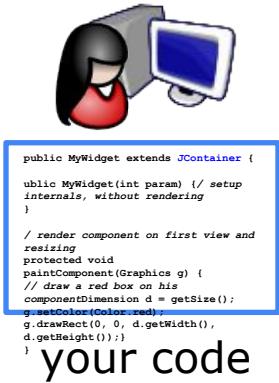
Using the example framework

```
public abstract class Application extends JFrame {  
    protected String getApplicationTitle() { return ""; }  
    protected String getButtonText() { return ""; }  
    protected String getInitialText() { return ""; }  
  
    public class Calculator extends Application {  
        protected String getApplicationTitle() { return "My Great Calculator"; }  
        protected String getButtonText() { return "calculate"; }  
        protected String getInitialText() { return "(10 - 3) * 6"; }  
        protected void buttonClicked() {  
            JOptionPane.showMessageDialog(this, "The result of " + getInput() +  
                " is " + calculate(getInput()));  
        }  
        private String calculate(String text) { ... }  
    }  
    textField.setPreferredSize(new Dimension(200, 20));  
    contentPane.add(textField, BorderLayout.WEST);  
    button.addActionListener((e) -> { buttonClicked(); });  
    this.setContentPane(contentPane);  
    this.pack();  
}
```

Using the example framework again

```
public abstract class Application extends JFrame {  
    protected String getApplicationTitle() { return ""; }  
    protected String getButtonText() { return ""; }  
    protected String getInitialText() { return ""; }  
  
    public class Calculator extends Application {  
        protected String getApplicationTitle() { return "My Great Calculator"; }  
        protected String getButtonText() { return "calculate"; }  
        protected String getInitialText() { return "(10 - 3) * 6"; }  
        protected void buttonClicked() {  
            JOptionPane.showMessageDialog(this, "The result of " + getInput() +  
                " is " + calculate(getInput()));  
        }  
        private String calculate(String text) {  
            ...  
        }  
    }  
}  
  
public class Ping extends Application {  
    protected String getApplicationTitle() { return "Ping"; }  
    protected String getButtonText() { return "ping"; }  
    protected String getInitialText() { return "127.0.0.1"; }  
    protected void buttonClicked() { ... }  
}
```

General distinction: Library vs. framework



Library

```
public MyWidget extends JPanel {  
    public MyWidget(int param) { /* setup  
        internals, without rendering  
    */  
  
        // render component on first view and  
        // resizing  
        protected void  
        paintComponent(Graphics g) {  
            // draw a red box on his  
            componentDimension d = getSize();  
            g.setColor(Color.red);  
            g.drawRect(0, 0, d.getWidth(),  
            d.getHeight());  
        }  
    }  
}  
your code
```

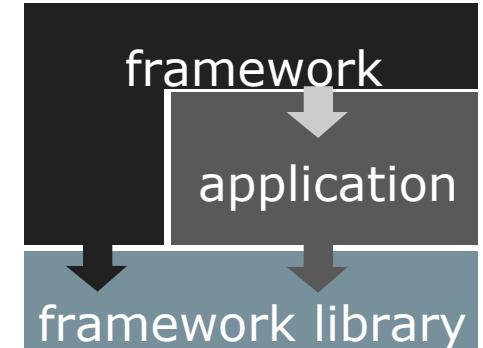
Framework



user
interacts

Libraries and frameworks in practice

- Defines key abstractions and their interfaces
- Defines object interactions & invariants
- Defines flow of control
- Provides architectural guidance
- **Provides defaults**



credit: Erich Gamma

Framework or library?

- IntelliJ / VSCode
- Java Collections / Node Streams

Framework or library?

- IntelliJ / VSCode
- Java Collections / Node Streams
- Command line parser
- Express/NanoHttpd
- Handlebars (the template library used in HW4)
- On a piece of paper:
 1. Describe the software (<= one sentence)
 2. Describe one way the software is like a library.
 3. Describe one way the software is like a framework.

Is Santorini a Framework?



More terms

- *API*: Application Programming Interface, the interface of a library or framework
- *Client*: The code that uses an API
- *Plugin*: Client code that customizes a framework
- *Extension point*: A place where a framework supports extension with a plugin

More terms

- *Protocol*: The expected sequence of interactions between the API and the client
- *Callback*: A plugin method that the framework will call to access customized functionality
- *Lifecycle method*: A callback method that gets called in a sequence according to the protocol and the state of the plugin

WHITE-BOX VS BLACK-BOX* **FRAMEWORKS**

* old terms, not aware of common replacements; maybe Inheritance-Based vs Delegation-Based Frameworks

Whitebox (inheritance-based) frameworks

- Extension via subclassing and overriding methods
- Common design pattern(s):
 - Template method
- Subclass has main method but gives control to framework

Blackbox (delegation-based) frameworks

- Extension via implementing a plugin interface
- Common design pattern(s):
 - Strategy
 - Command
 - Observer
- Plugin-loading mechanism loads plugins and gives control to the framework

Is this a whitebox or blackbox framework?

```
public abstract class Application extends JFrame {  
    protected String getApplicationTitle() { return ""; }  
    protected String getButtonText() { return ""; }  
    protected String getInitialText() { return ""; }  
  
    public class Calculator extends Application {  
        protected String getApplicationTitle() { return "My Great Calculator"; }  
        protected String getButtonText() { return "calculate"; }  
        protected String getInitialText() { return "(10 - 3) * 6"; }  
        protected void buttonClicked() {  
            JOptionPane.showMessageDialog(this, "The result of " + getInput() +  
                " is " + calculate(getInput()));  
        }  
  
        public class Ping extends Application {  
            protected String getApplicationTitle() { return "Ping"; }  
            protected String getButtonText() { return "ping"; }  
            protected String getInitialText() { return "127.0.0.1"; }  
            protected void buttonClicked() { ... }  
        }  
    }  
}
```

An example blackbox framework

```
public class Application extends JFrame {  
    private JTextField textField;  
    private Plugin plugin;  
    public Application() { }  
    protected void init(Plugin p) {  
        p.setApplication(this);  
        this.plugin = p;  
        JPanel contentPane = new JPanel();  
        contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));  
        JButton button = new JButton();  
        button.setText(plugin != null ? plugin.getButtonText() : "ok");  
        contentPane.add(button, BorderLayout.EAST);  
        textField = new JTextField("");  
        if (plugin != null) textField.setText(plugin.getInitialText());  
        textField.setPreferredSize(new Dimension(200, 20));  
        contentPane.add(textField, BorderLayout.WEST);  
        if (plugin != null)  
            button.addActionListener((e) -> { plugin.buttonClicked(); } );  
    this.setContentPane(contentPane);  
}
```

```
public interface Plugin {  
    String getApplicationTitle();  
    String getButtonText();  
    String getInitialText();  
    void buttonClicked();  
    void setApplication(Application app);  
}
```

An example blackbox framework

```
public class Application extends JFrame {  
    private JTextField textField;  
    private Plugin plugin;  
    public Application() { }  
    protected void init(Plugin p) {  
        p.setApplication(this);  
        this.plugin = p;  
    }  
  
    public class CalcPlugin implements Plugin {  
        private Application app;  
        public void setApplication(Application 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 Plugin {  
    String getApplicationTitle();  
    String getButtonText();  
    String getInititalText();  
    void buttonClicked();  
    void setApplication(Application app);  
}
```

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 getInitialText();  
    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 getInitialText() { 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();  
}
```

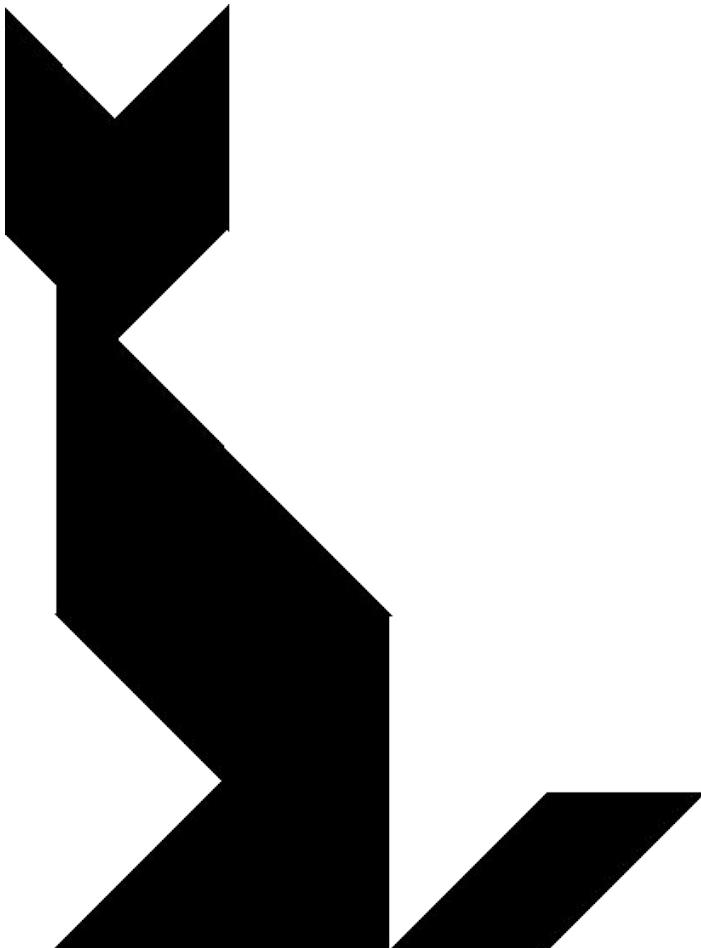
Framework summary

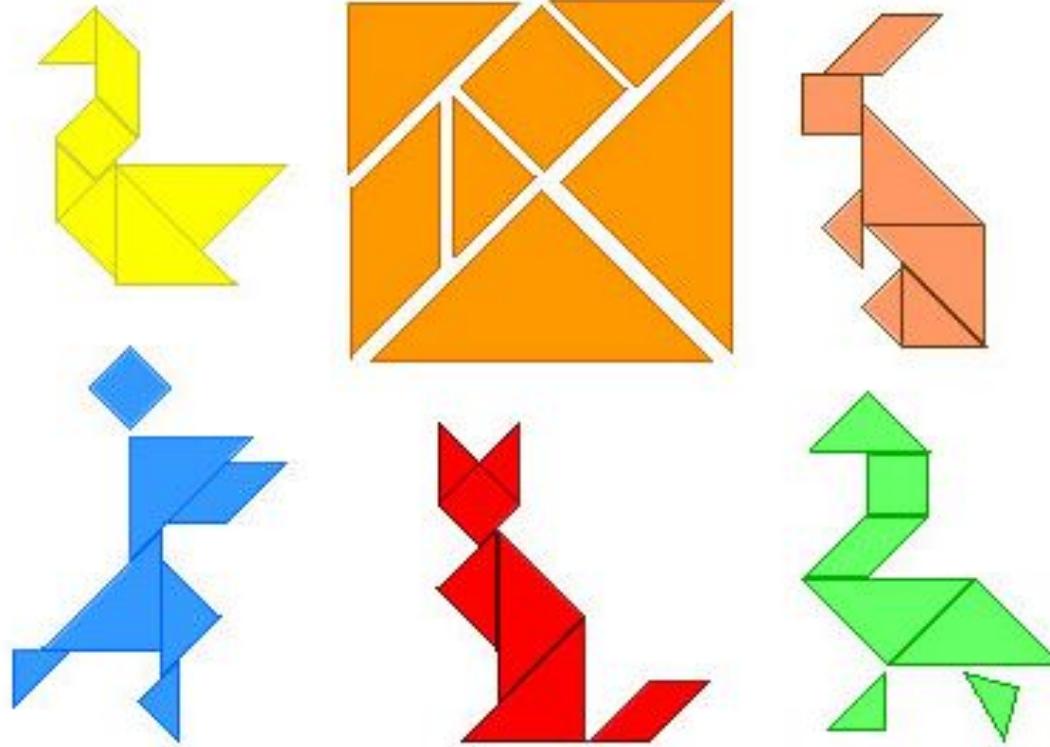
- Whitebox frameworks use subclassing
 - Allows extension of every nonprivate method
 - Need to understand implementation of superclass
 - Only one extension at a time
 - Compiled together
 - Often so-called developer frameworks
- Blackbox frameworks use composition
 - Allows extension of functionality exposed in interface
 - Only need to understand the interface
 - Multiple plugins
 - Often provides more modularity
 - Separate deployment possible (.jar, .dll, ...)
 - Often so-called end-user frameworks, platforms

Framework design considerations

- Once designed there is little opportunity for change
- Key decision: Separating common parts from variable parts
 - What problems do you want to solve?
- Possible problems:
 - Too few extension points: Limited to a narrow class of users
 - Too many extension points: Hard to learn, slow
 - Too generic: Little reuse value

USE VS REUSE: DOMAIN ENGINEERING





(one modularization: tangrams)

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

Domain engineering

- Understand users/customers in your domain: What might they need? What extensions are likely?
- Collect example applications before designing a framework
- Make a conscious decision what to support (*scoping*)
- e.g., the Eclipse policy:
 - Plugin interfaces are internal at first
 - Unsupported, may change
 - Public stable extension points created when there are at least two distinct customers

The cost of changing a framework

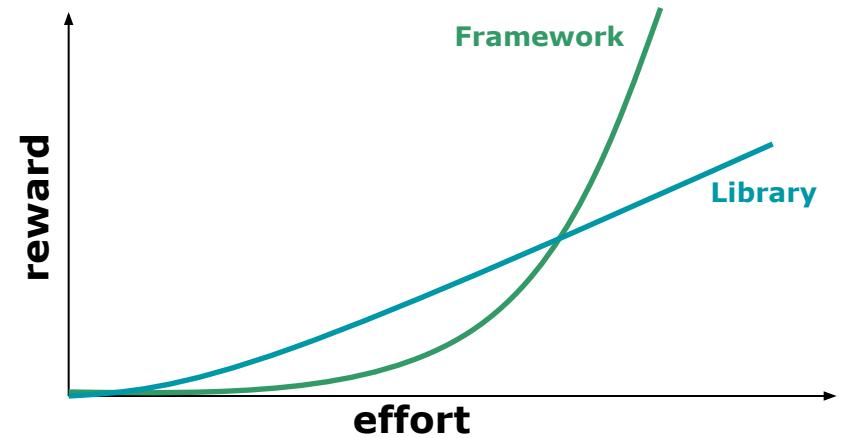
```
public class Application extends JFrame {  
    private JTextField textfield;  
    private Plugin plugin;  
    public Application(Plugin p) { this.plugin=p; p.setApplication(this); init(); }  
    protected void init() {  
        JPanel contentPane = new JPanel(new BorderLayout());  
        contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));  
        JButton button = new JButton();  
        if (plugin != null)  
            button.setText(plugin.getButtonText());  
        else  
            button.setText("Calculate");  
        contentPane.add(button, "Center");  
        textfield = new JTextField("0");  
        contentPane.add(textfield, "South");  
        if (plugin != null)  
            textfield.setText(plugin.getInitialText());  
    }  
  
    public interface Plugin {  
        String getApplicationTitle();  
        String getButtonText();  
        String getInitialText();  
        void buttonClicked();  
        void setApplication(Application app);  
    }  
  
    public class CalcPlugin implements Plugin {  
        private Application application;  
        public void setApplication(Application app) { this.application = app; }  
        public String getButtonText() { return "calculate"; }  
        public String getInitialText() { return "10 / 2 + 6"; }  
        public void buttonClicked() {  
            String result = application.getText();  
            int value = Integer.parseInt(result);  
            result = String.valueOf(value);  
            application.setText(result);  
        }  
    }  
}
```

Consider adding an extra method.
Many changes require changes to *all* plugins.

```
l, "The result of "  
() + " is "  
n.getText())); }  
) { return "My Great Calculator"; }
```

Learning a framework

- Documentation
- Tutorials, wizards, and examples
- Communities, email lists and forums
- Other client applications and plugins



Typical framework design and implementation

Define your domain

Identify potential common parts and variable parts

Design and write sample plugins/applications

Factor out & implement common parts as framework

Provide plugin interface & callback mechanisms for variable parts

Use well-known design principles and patterns where appropriate...

Get lots of feedback, and iterate

FRAMEWORK MECHANICS

Running a framework

- Some frameworks are runnable by themselves
 - e.g. Eclipse, VSCode, IntelliJ
- Other frameworks must be extended to be run
 - MapReduce, Swing, JUnit, NanoHttpd, Express

Methods to load plugins

Client writes main function, creates a plugin object, and passes it to framework
(see blackbox example above)

Framework has main function, client passes name of plugin as a command line argument or environment variable
(see next slide)

Framework looks in a magic location

Config files or .jar/.js files in a plugins/ directory are automatically loaded and processed

GUI for plugin management

An example plugin loader using Java Reflection

```
public static void main(String[] args) {
    if (args.length != 1)
        System.out.println("Plugin name not specified");
    else {
        String pluginName = args[0];
        try {
            Class<?> pluginClass = Class.forName(pluginName);
            new Application((Plugin) pluginClass.newInstance()).setVisible(true);
        } catch (Exception e) {
            System.out.println("Cannot load plugin " + pluginName
                + ", reason: " + e);
        }
    }
}
```

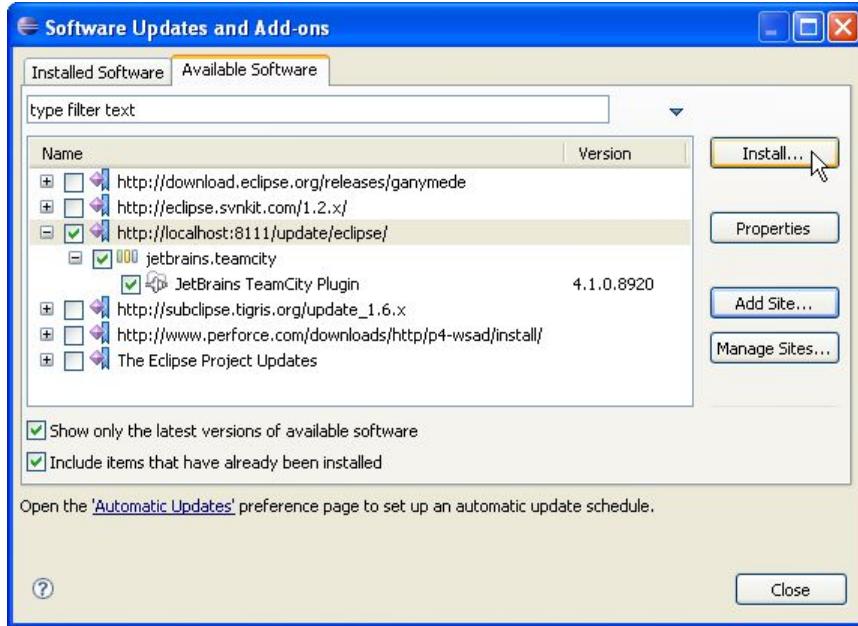
An example plugin loader in Node.js

```
const args = process.argv
if (args.length < 3)
    console.log("Plugin name not specified");
else {
    const plugin = require("plugins/"+args[2]+".js")()
    startApplication(plugin)
}
```

Another plugin loader using Java Reflection

```
public static void main(String[] args) {
    File config = new File(".config");
    BufferedReader reader = new BufferedReader(new FileReader(config));
    Application application = new Application();
    Line line = null;
    while ((line = reader.readLine()) != null) {
        try {
            Class<?> pluginClass = Class.forName(pluginName);
            application.addPlugin((Plugin) pluginClass.newInstance());
        } catch (Exception e) {
            System.out.println("Cannot load plugin " + pluginName
                + ", reason: " + e);
        }
    }
    reader.close();
    application.setVisible(true);
}
```

GUI-based plugin management



Supporting multiple plugins

- Observer design pattern is commonly used
- Load and initialize multiple plugins
- Plugins can register for events
- Multiple plugins can react to same events
- Different interfaces for different events possible

```
public class Application {  
    private List<Plugin> plugins;  
    public Application(List<Plugin> plugins) {  
        this.plugins=plugins;  
        for (Plugin plugin: plugins)  
            plugin.setApplication(this);  
    }  
    public Message processMsg (Message msg) {  
        for (Plugin plugin: plugins)  
            msg = plugin.process(msg);  
        ...  
        return msg;  
    }  
}
```

Example: An Eclipse plugin

- A popular Java IDE
- More generally, a framework for tools that facilitate “building, deploying and managing software across the lifecycle.”
- Plugin framework based on OSGI standard
- Starting point: Manifest file
 - Plugin name
 - Activator class
 - Meta-data

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: MyEditor Plug-in
Bundle-SymbolicName: MyEditor;
singleton:=true
Bundle-Version: 1.0.0
Bundle-Activator:
myeditor.Activator
Require-Bundle:
org.eclipse.ui,
org.eclipse.core.runtime,
org.eclipse.jface.text,
org.eclipse.ui.editors
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment:
JavaSE-1.6
```

Example: An Eclipse plugin

- plugin.xml
 - Main configuration file
 - XML format
 - Lists extension points
- Editor extension
 - extension point: org.eclipse.ui.editors
 - file extension
 - icon used in corner of editor
 - class name
 - unique id
 - refer to this editor
 - other plugins can extend with new menu items, etc.!

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>

<extension
    point="org.eclipse.ui.editors">
    <editor
        name="Sample XML Editor"
        extensions="xml"
        icon="icons/sample.gif"
        contributorClass="org.eclipse.ui.texteditor.BasicText
        EditorActionContributor"
        class="myeditor.editors.XMLEditor"
        id="myeditor.editors.XMLEditor">
        </editor>
    </extension>

</plugin>
```

Example: An Eclipse plugin

- At last, code!
- XMLEditor.java
 - Inherits TextEditor behavior
 - open, close, save, display, select, cut/copy/paste, search/replace, ...
 - REALLY NICE not to have to implement this
 - But could have used ITextEditor interface if we wanted to
 - Extends with syntax highlighting
 - XMLDocumentProvider partitions into tags and comments
 - XMLConfiguration shows how to color partitions

```
package myeditor.editors;

import org.eclipse.ui.editors.text.TextEditor;

public class XMLEditor extends TextEditor {
    private ColorManager colorManager;

    public XMLEditor() {
        super();
        colorManager = new
            ColorManager();
        setSourceViewerConfiguration(
            new XMLConfiguration(colorManager));
        setDocumentProvider(
            new XMLDocumentProvider());
    }

    public void dispose() {
        colorManager.dispose();
        super.dispose();
    }
}
```

Example: A JUnit Plugin

```
public class SampleTest {  
    private List<String> emptyList;  
  
    @Before  
    public void setUp() {  
        emptyList = new ArrayList<String>();  
    }  
  
    @After  
    public void tearDown() {  
        emptyList = null;  
    }  
  
    @Test  
    public void testEmptyList() {  
        assertEquals("Empty list should have 0 elements",  
                    0, emptyList.size());  
    }  
}
```

Here the important plugin mechanism is Java annotations

Summary

- Reuse and variation essential
 - Libraries and frameworks
- Whitebox frameworks vs. blackbox frameworks
- Design for reuse with domain analysis
 - Find common and variable parts
 - Write client applications to find common parts