Principles of Software Construction: Objects, Design, and Concurrency

Asynchrony and Concurrency

Christian Kästner **Vincent Hellendoorn**

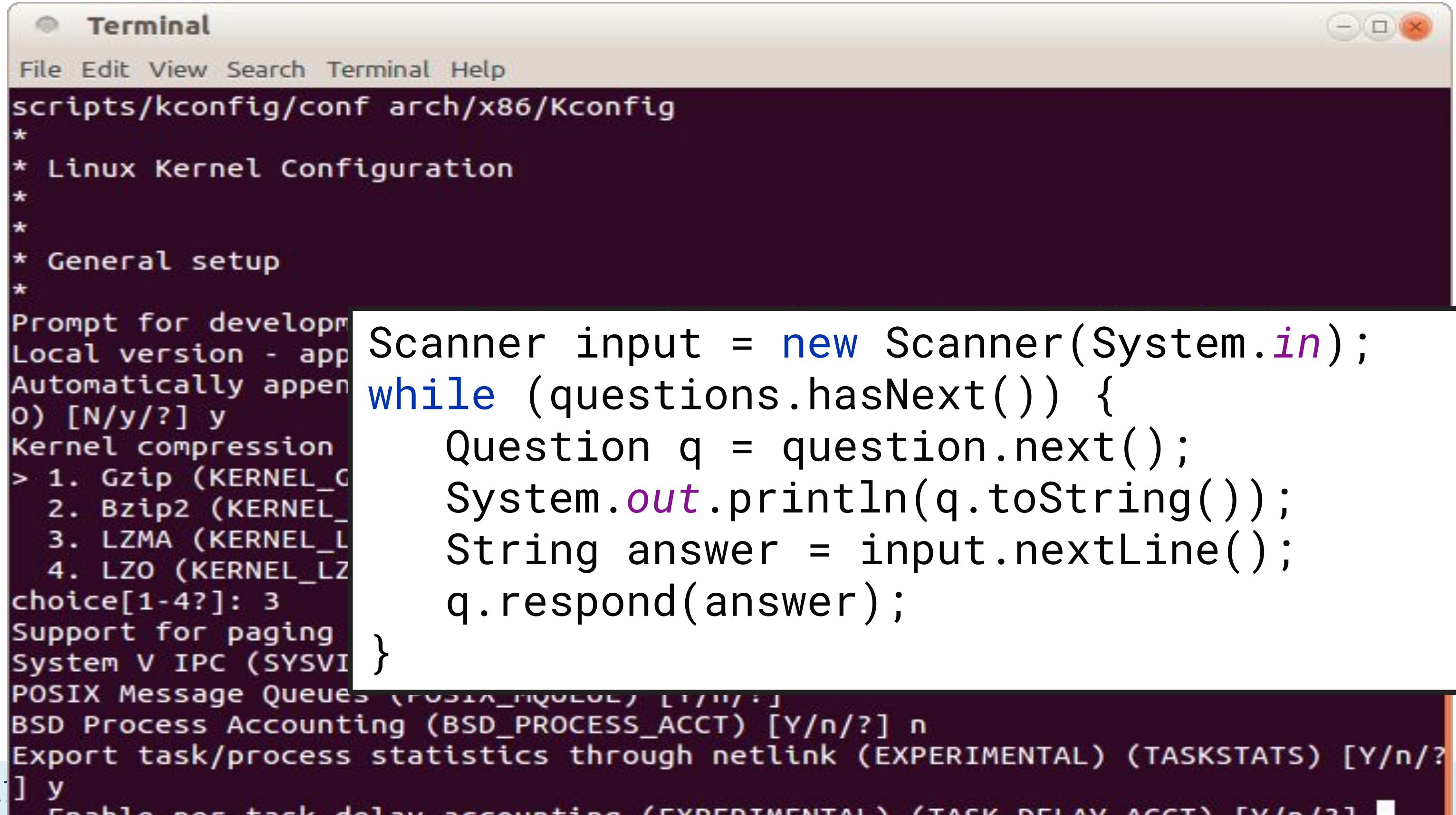
Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

How was the Recitation?

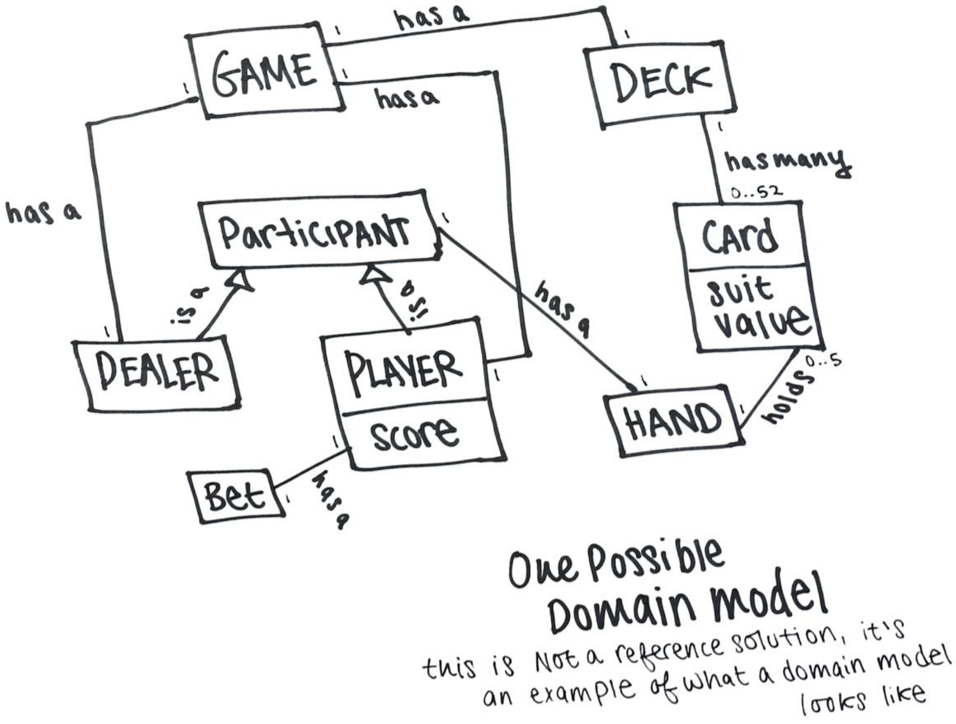
- Did every solution make the program smaller?
- Did I change everything you would have?
  - Anything you wouldn’t?

Interaction with CLI



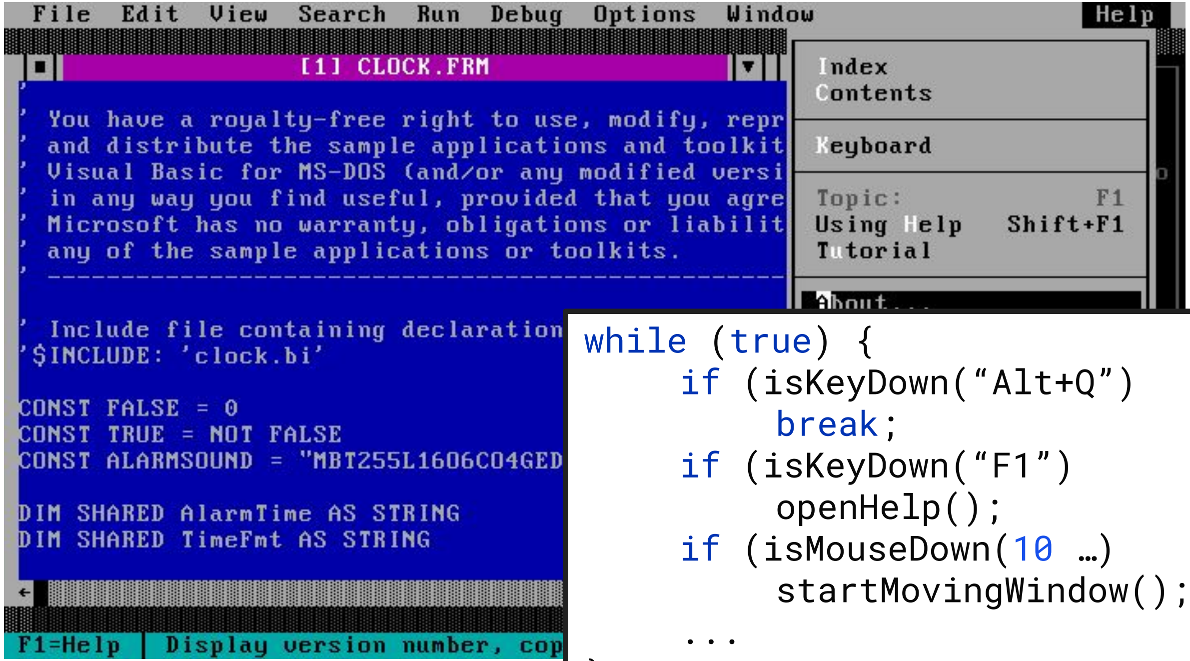

```java
Scanner input = new Scanner(System.in);
while (questions.hasNext()) {
   Question q = question.next();
   System.out.println(q.toString());
   String answer = input.nextLine();
   q.respond(answer);
}
```

A backend with no interaction



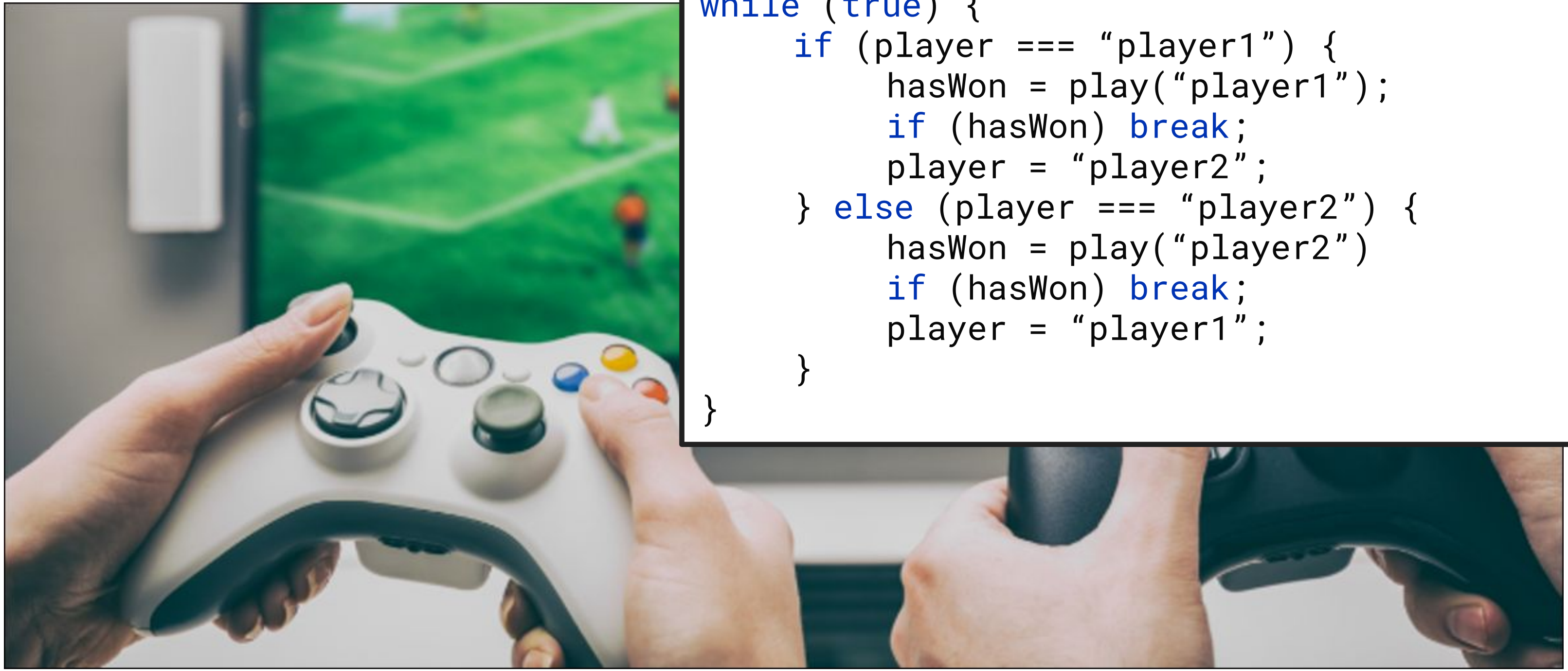
What have we not yet seen?

How do you <u>wait</u>?




```
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”;
    }
}
```

https://www.cloudsavvyit.com/2586/how-to-build-your-multiplayer-games-server-architecture/

Today

Beyond serial execution

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

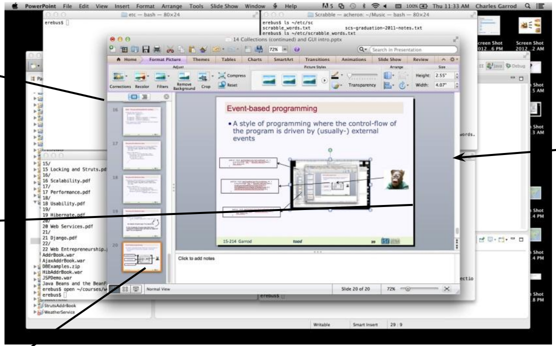
Event-based programming

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

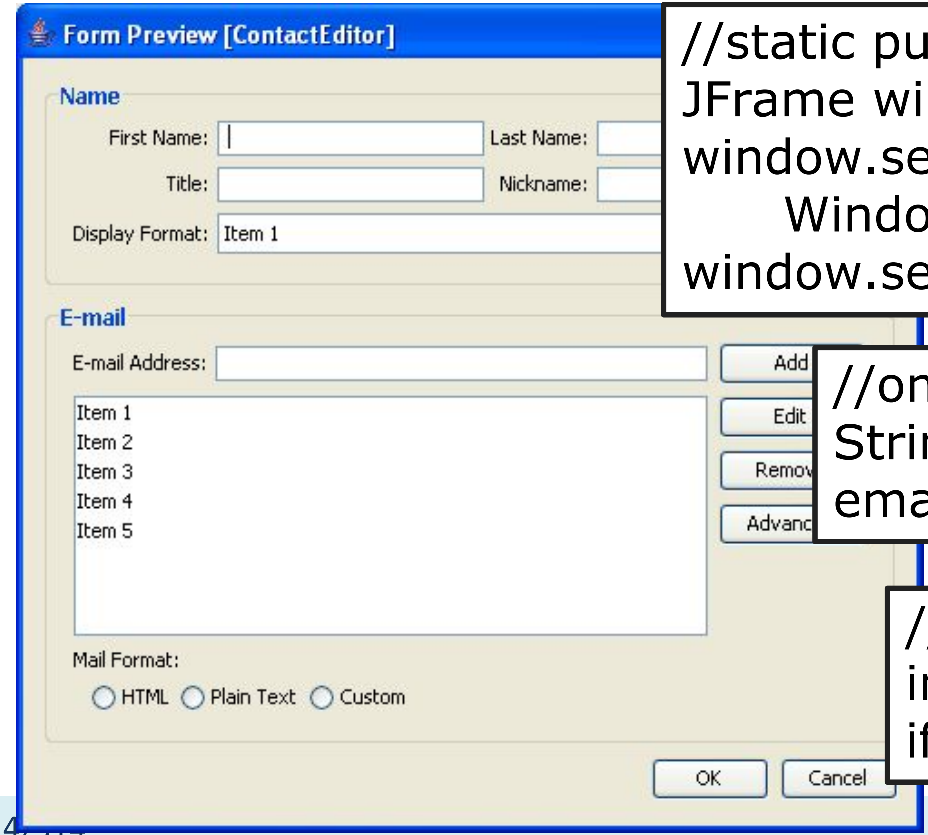
```
public void performAction(ActionEvent e) {
    List<String> lst = Arrays.asList(bar);
    foo.peek(42)
}
```

```
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

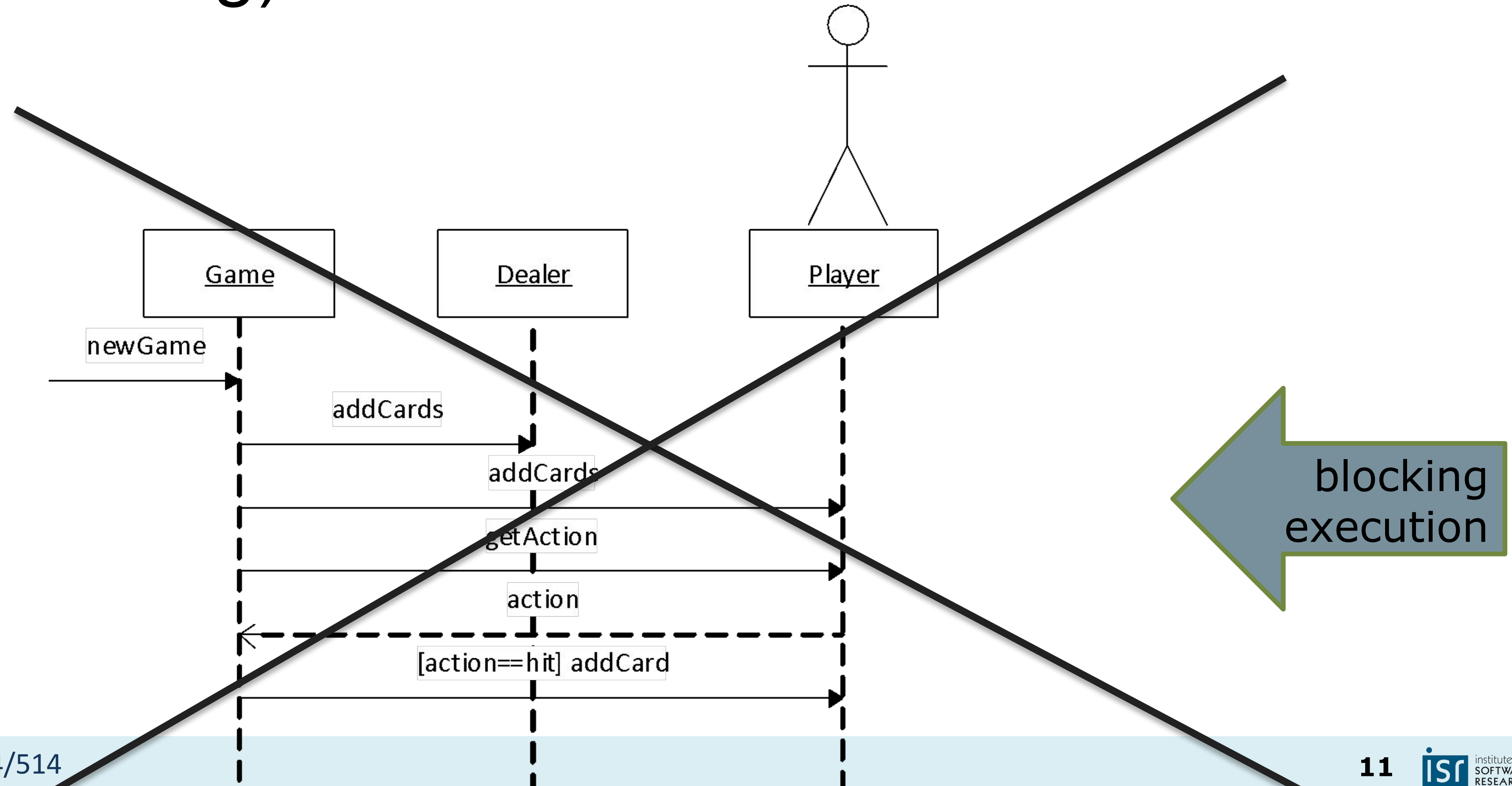



```
//static public void main…
JFrame window = …
window.setDefaultCloseOperation(
     WindowConstants.EXIT_ON_CLOSE);
window.setVisible(true);
```

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

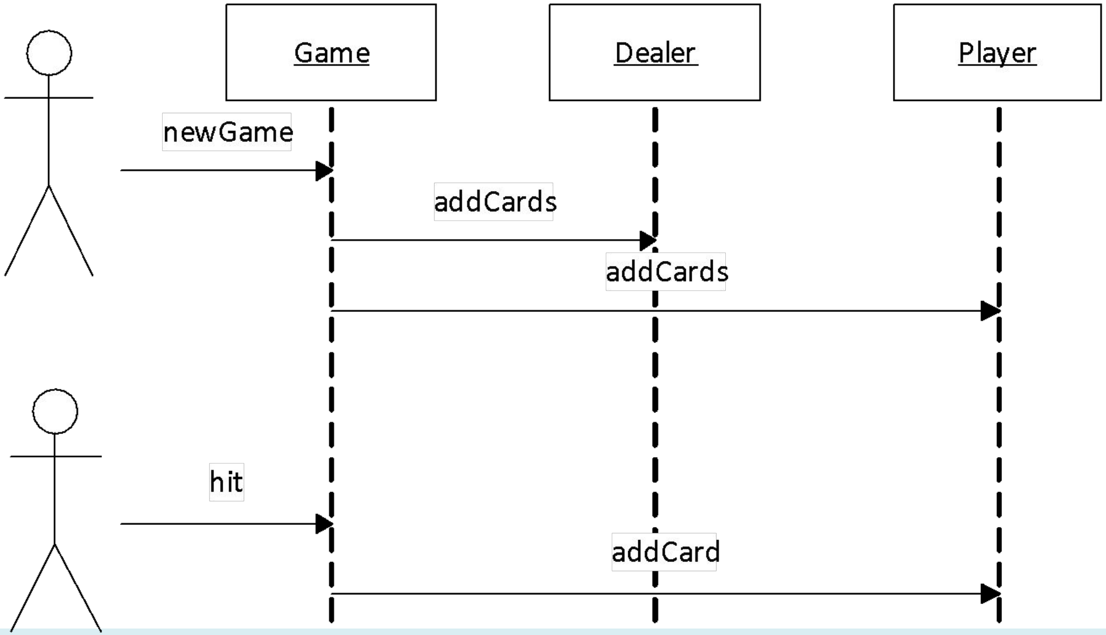
```
//on remove-button click:
int pos =emaillist.getSelectedItem(
if (pos>=0) emaillist.delete(pos);
```

(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?

institute for SOFTWARE RESEARCH

Asynchrony

Where might this come from?

- People
- Other machines
- Our own *callbacks*

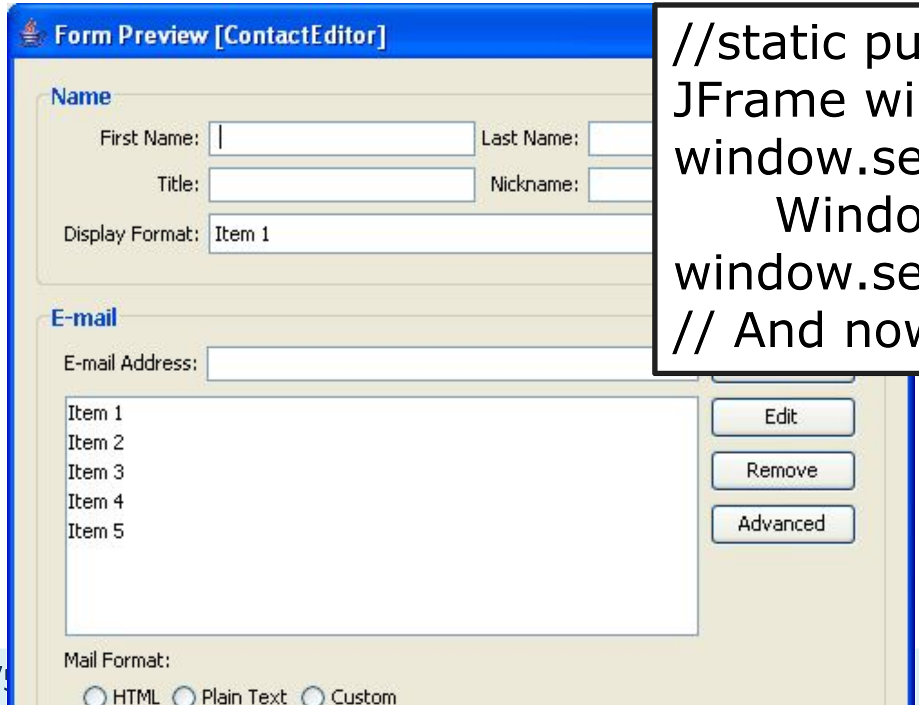
Asynchrony

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 Terms [i]

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

Concurrency with file I/O

Mostly used <u>synchronous</u> 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

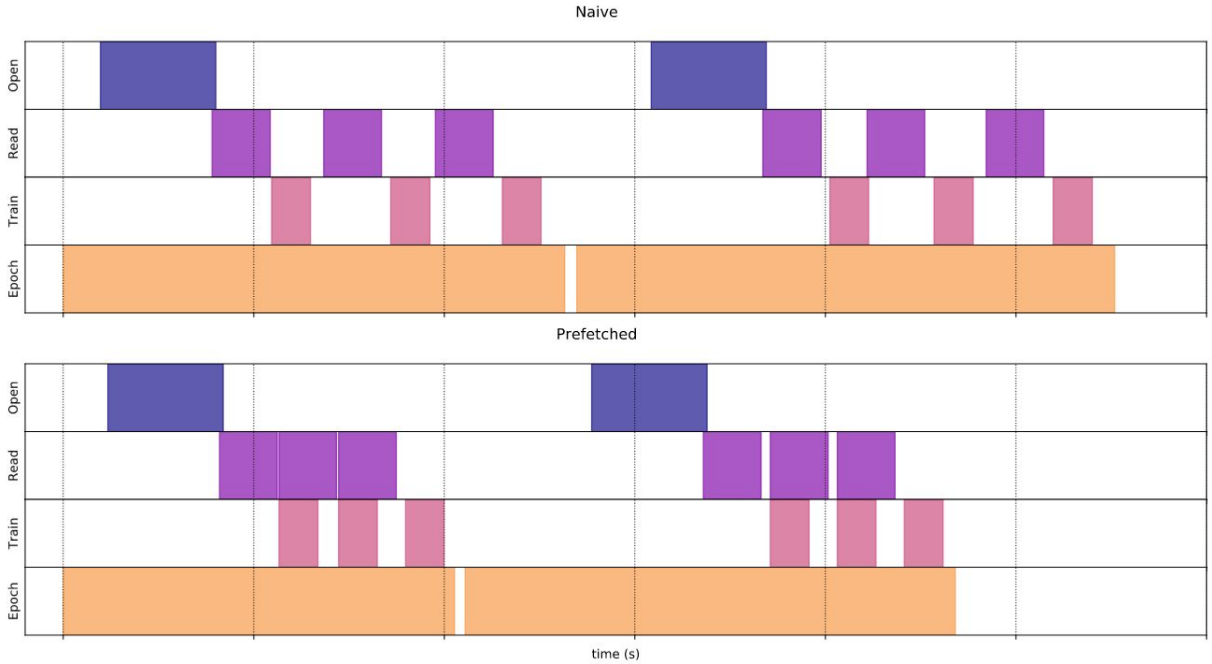
Can save you a lot of time

- 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

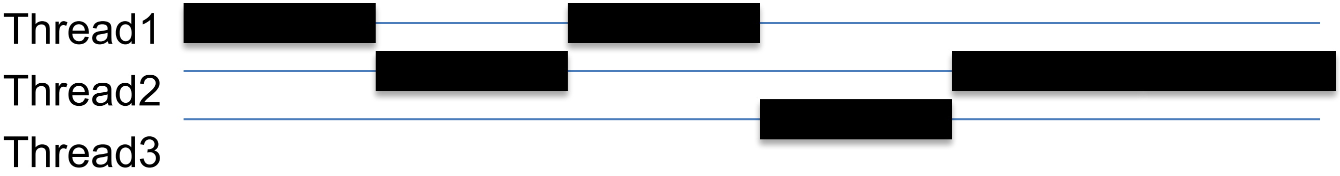
An example from Machine Learning

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

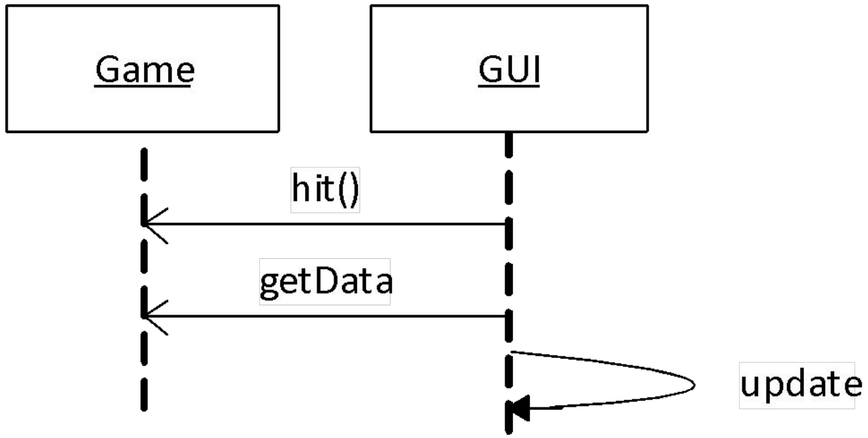
- Synchronization
- Immutability
- Parallelism
- More later in the course
  - Except for parallelism; largely out of scope

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

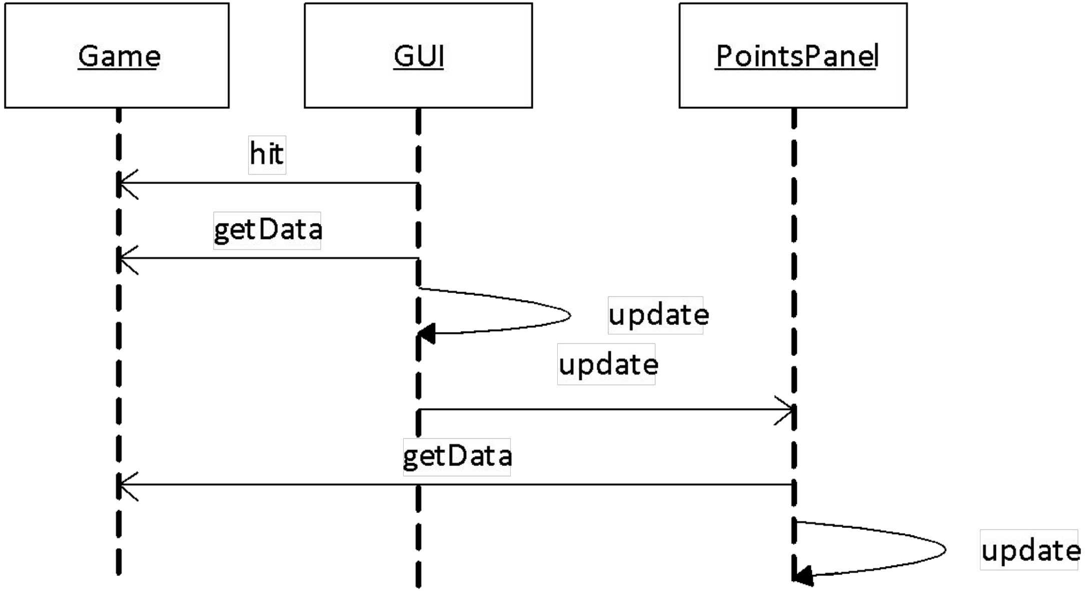
A GUI design challenge

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



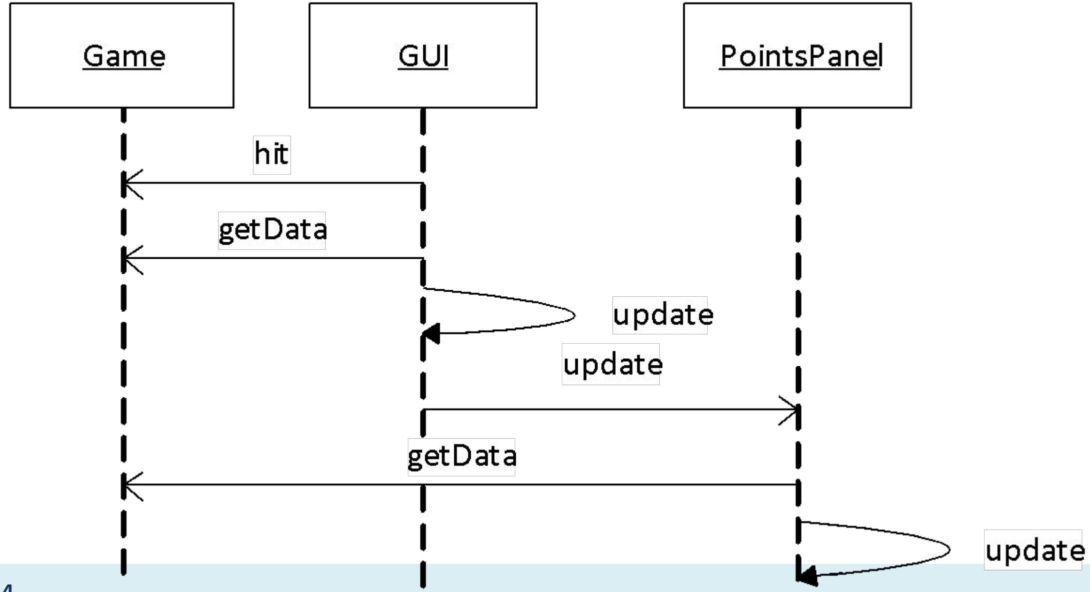
A GUI design challenge, extended

- What if we want to show the points won?



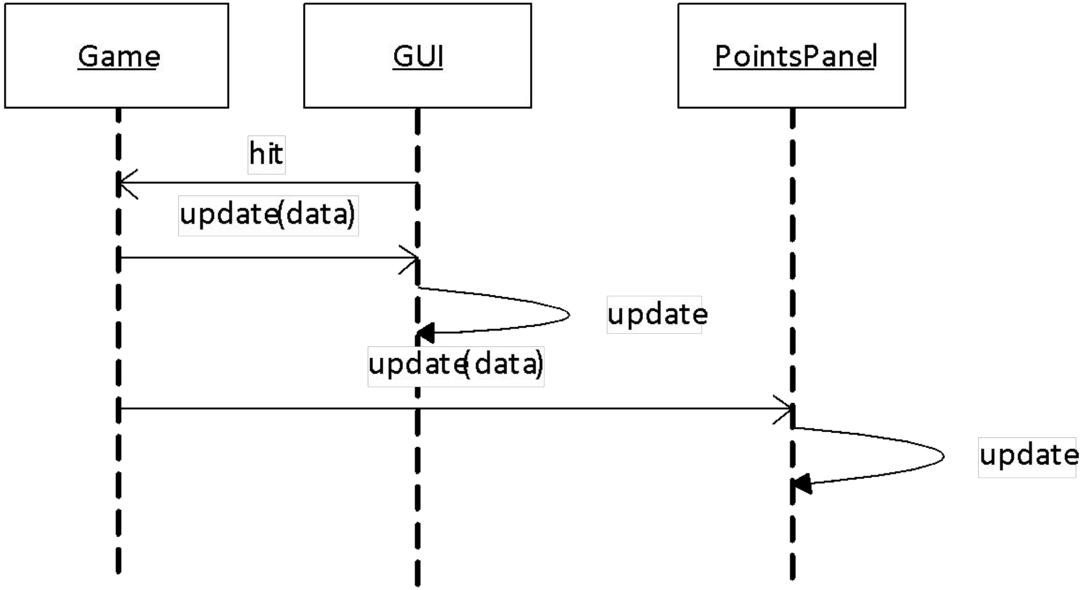
Game updates GUI?

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



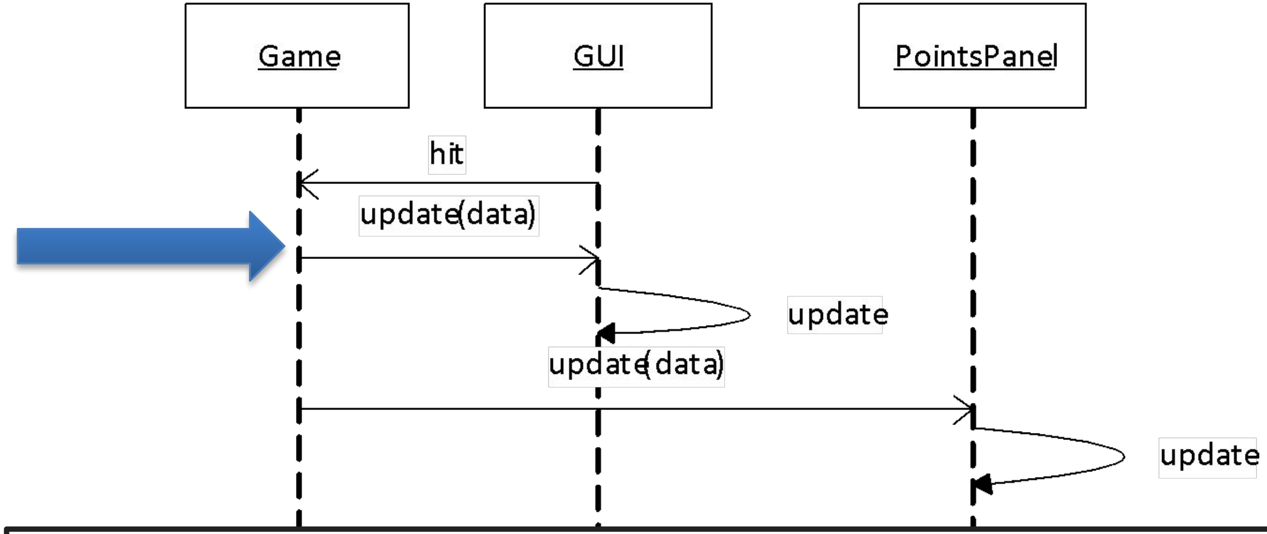
Game updates GUI?

- Let the Game tell the GUI that something happened



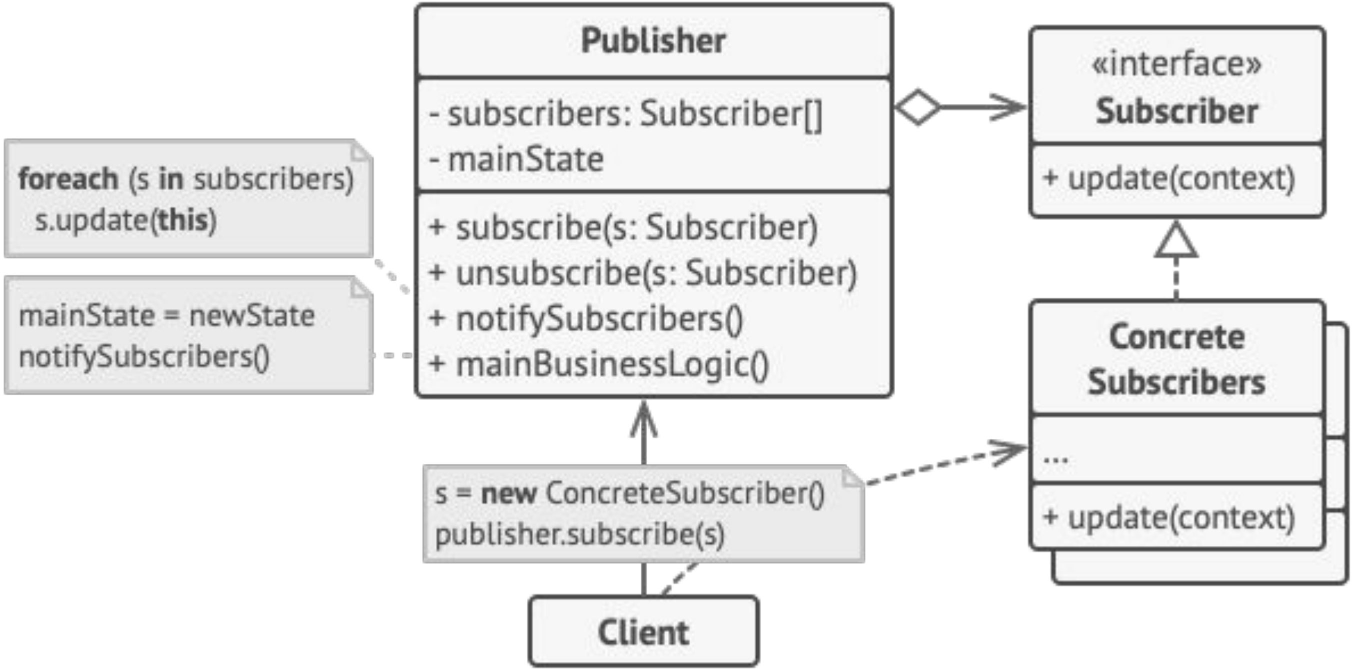
Game updates GUI?

- Let the Game tell the GUI that something happened



Problem: This couples the `World` to the GUI implementation.

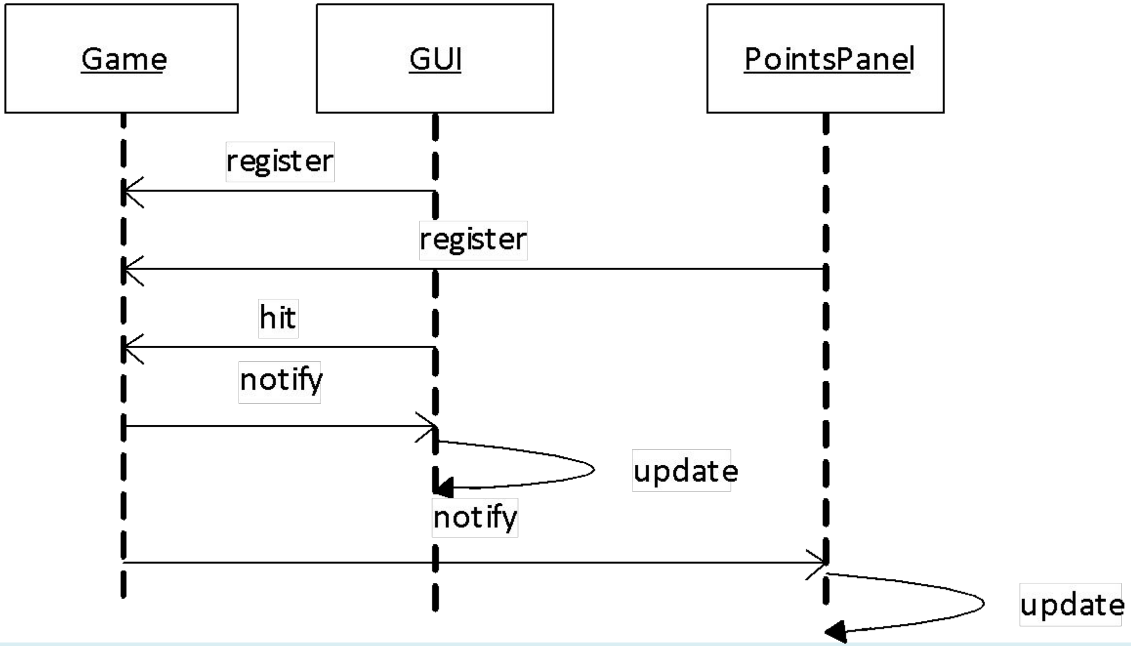
Recall the Observer



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

Decoupling with the Observer pattern

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

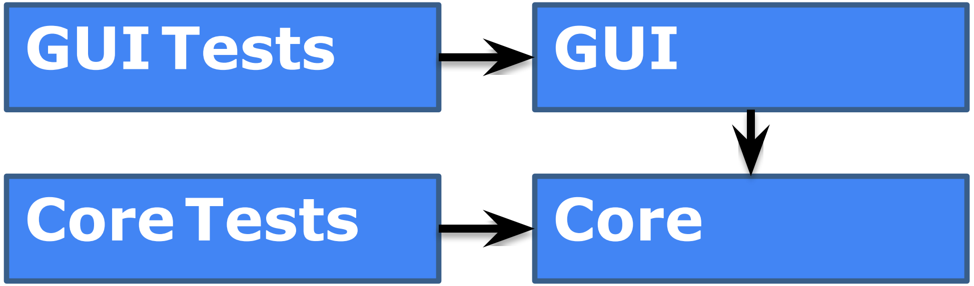


Core implementation vs. GUI

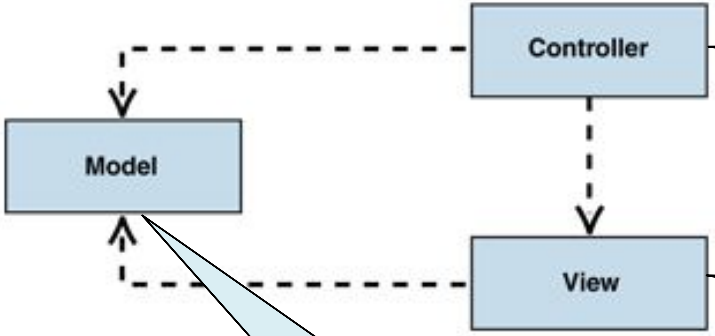
- Core implementation: application logic
  - Computing some result, updating data
- GUI
  - Graphical representation of data
  - Source of user interactions
- Design guideline: *avoid coupling the GUI with core application*
  - Multiple UIs with single core implementation
  - Test core without UI

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)



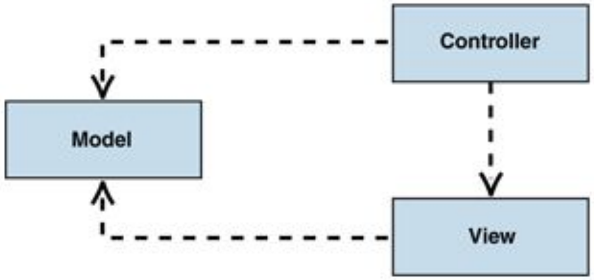
Manage inputs from user: mouse, keyboard, menu, etc.

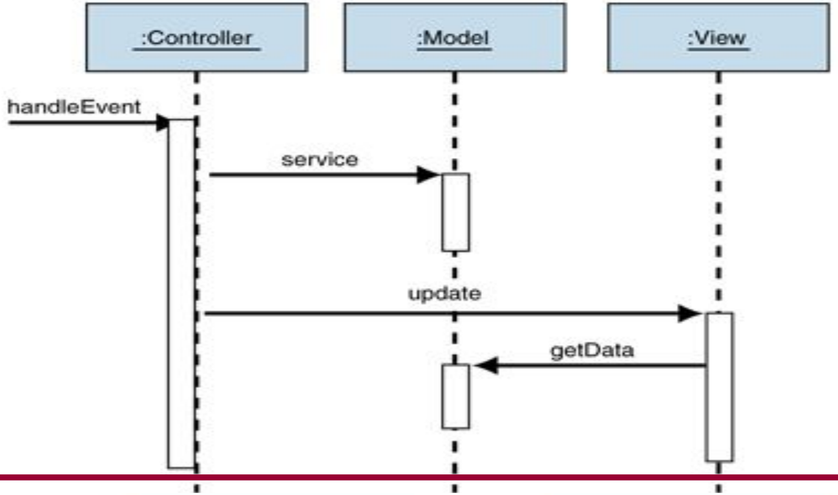
Manage display of information on the screen

Manage data related to the application domain

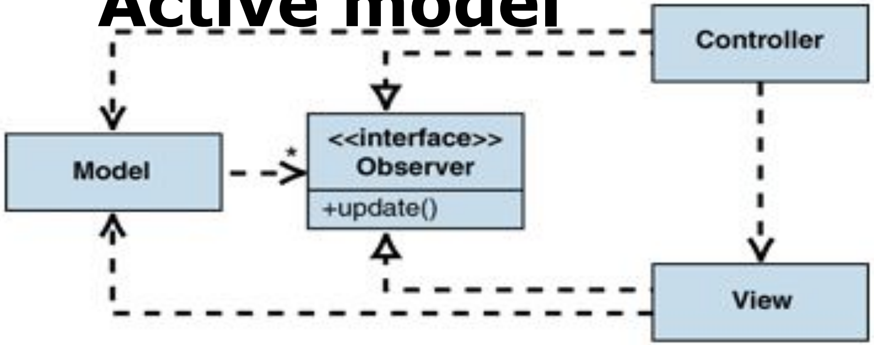
Model-View-Controller (MVC)

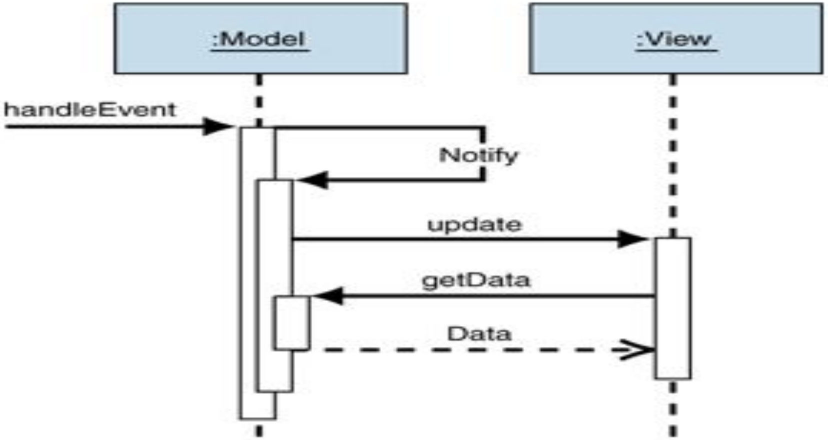
Passive model





Active model



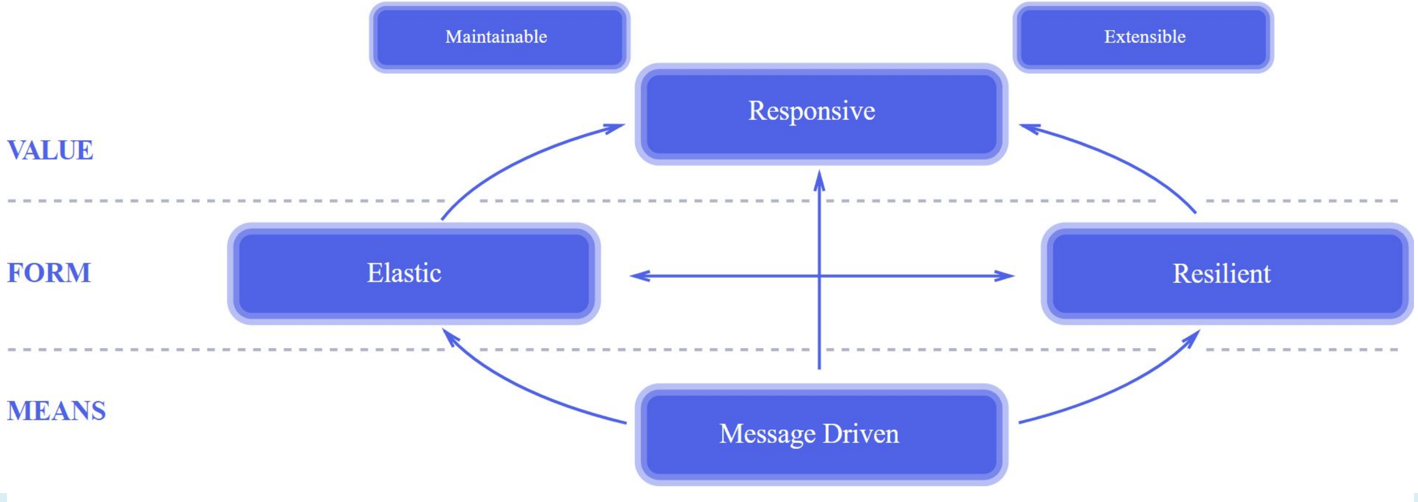


http://msdn.microsoft.com/en-us/library/ff649643.aspx

React Preview

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

- Without overwhelming workers
- Or blocking, or wasting resources



https://www.reactivemanifesto.org/

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