Principles of Software Construction: Objects, Design, and Concurrency

Asynchrony and Concurrency (leftovers)

Claire Le Goues **Bogdan Vasilescu**

Carnegie Mellon University
School of Computer Science

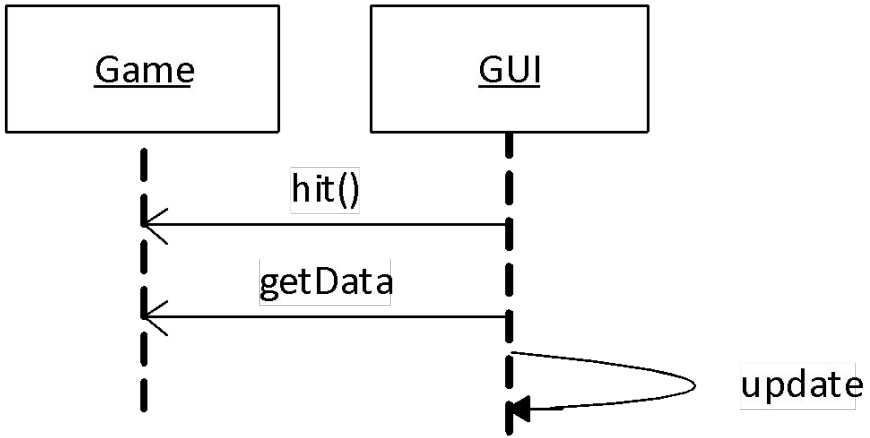
institute for SOFTWARE RESEARCH

A design challenge

DECOUPLING THE GUI

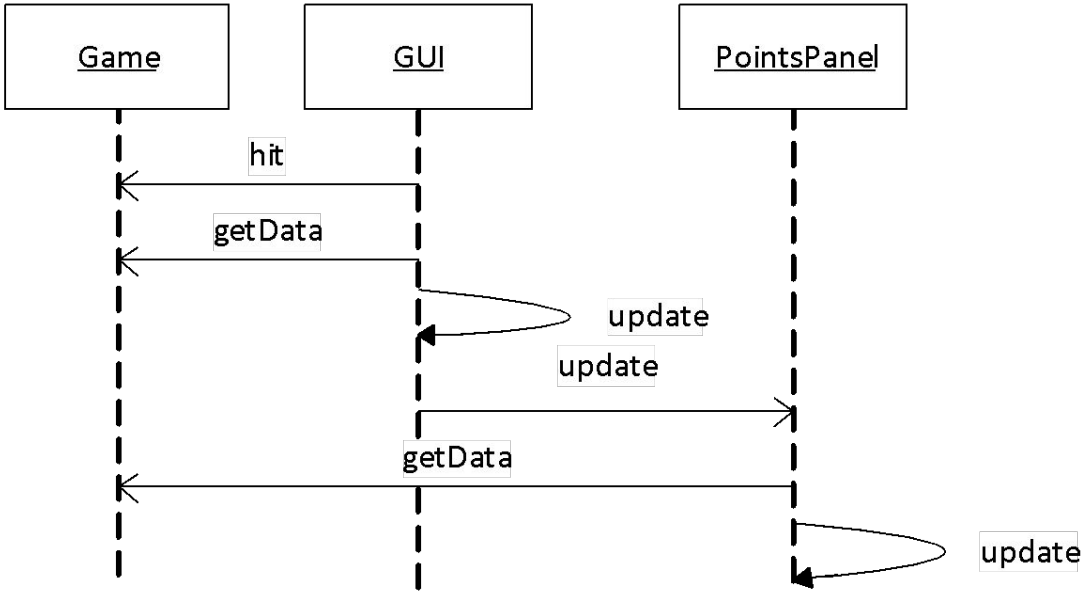
A GUI design challenge

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



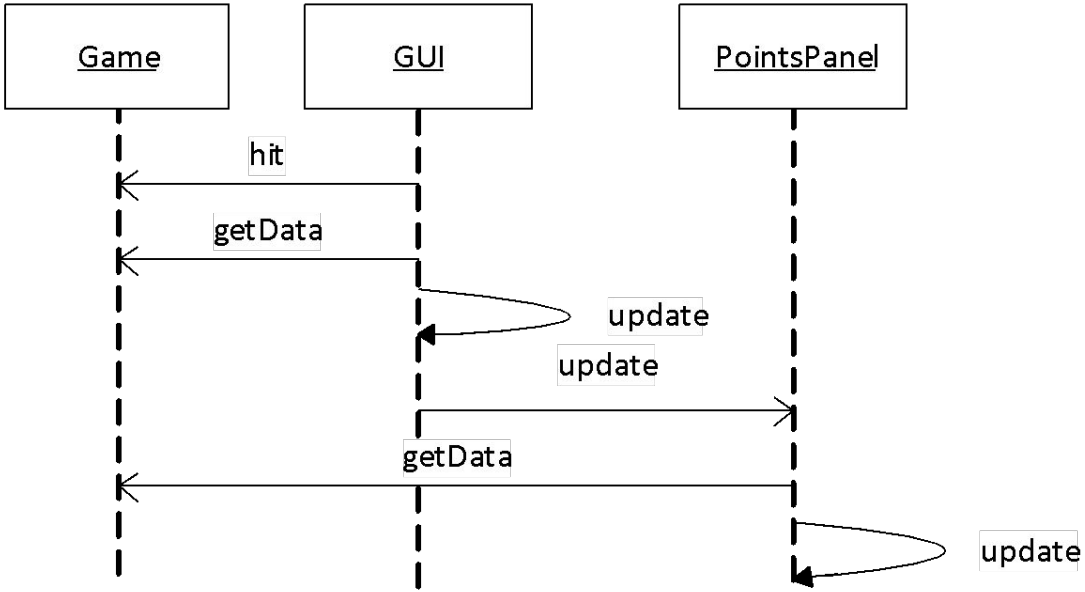
A GUI design challenge, extended

- What if we want to show the points won?



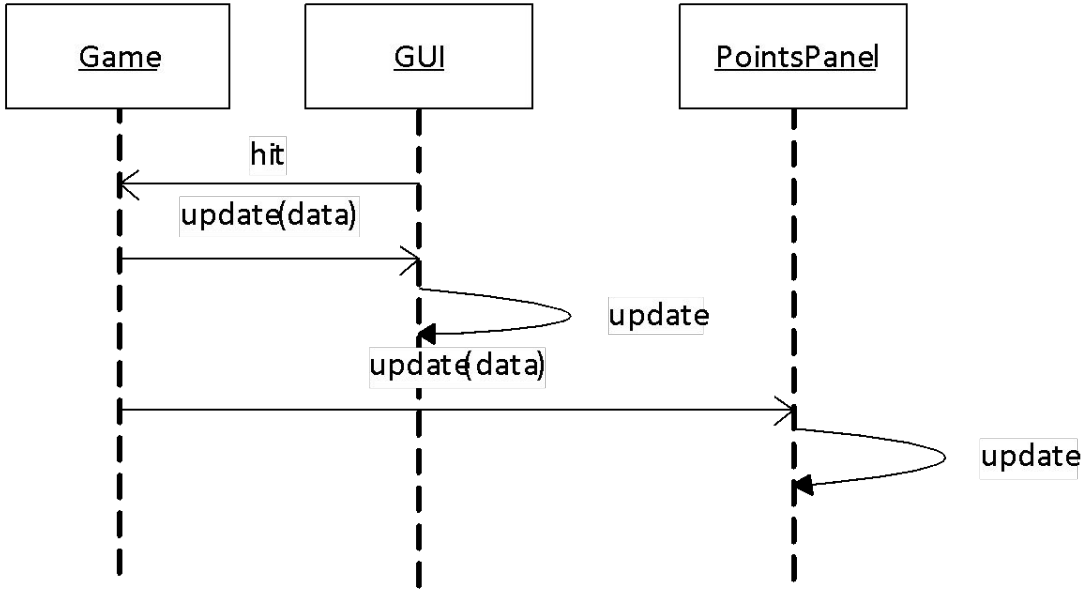
Game updates GUI?

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



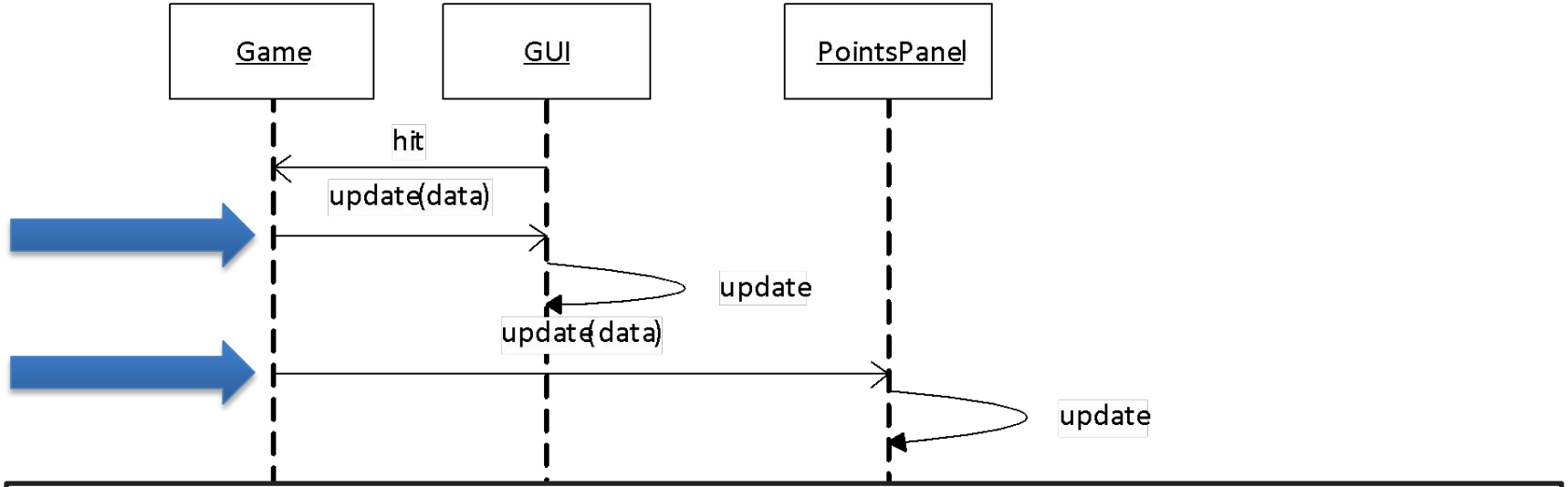
Game updates GUI?

- Let the Game tell the GUI that something happened



Game updates GUI?

- Let the Game tell the GUI that something happened



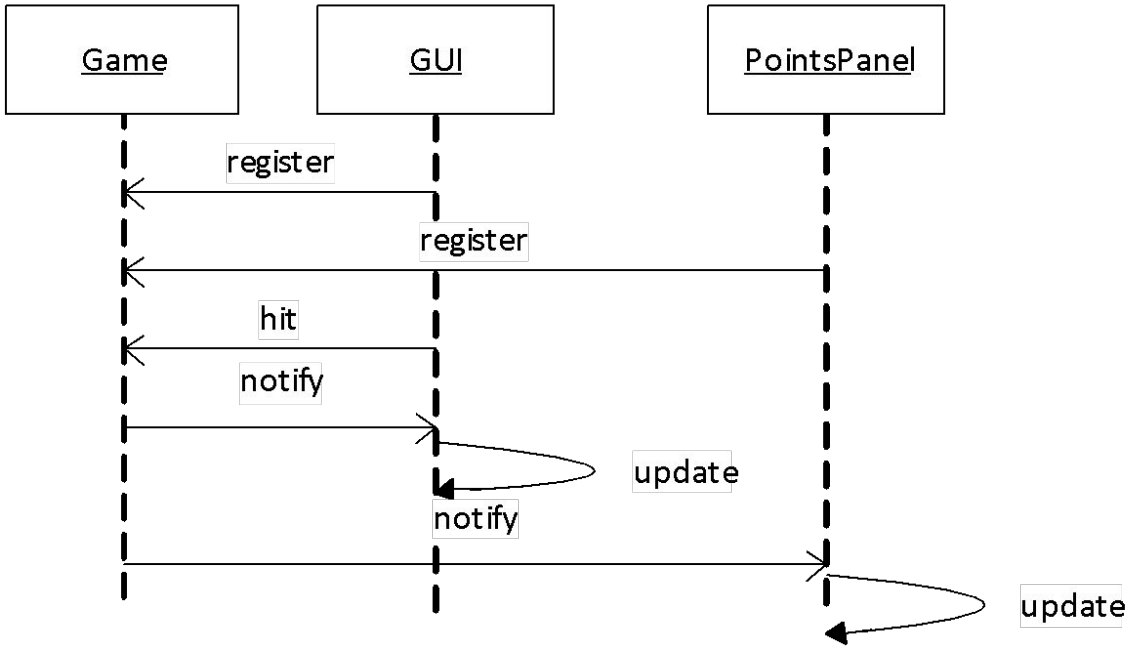
Problem: This couples the `World` to the GUI implementation.

Core implementation vs. GUI

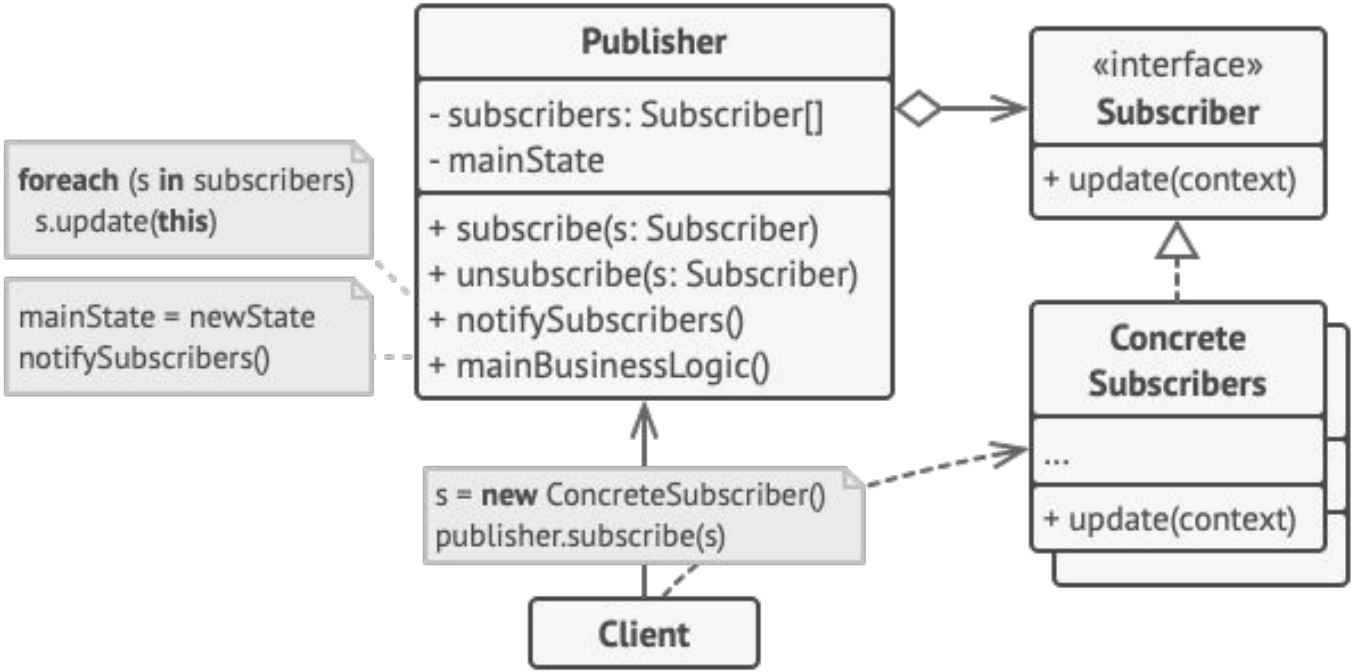
- Core implementation: Application logic
  - Computing some result, updating data
- GUI
  - Graphical representation of data
  - Source of user interactions
- Design guideline: *Avoid coupling the GUI with core application*
  - Multiple UIs with single core implementation
  - Test core without UI
  - *Design for change, design for reuse, design for division of labor; low coupling, high cohesion*

Decoupling with the Observer pattern

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



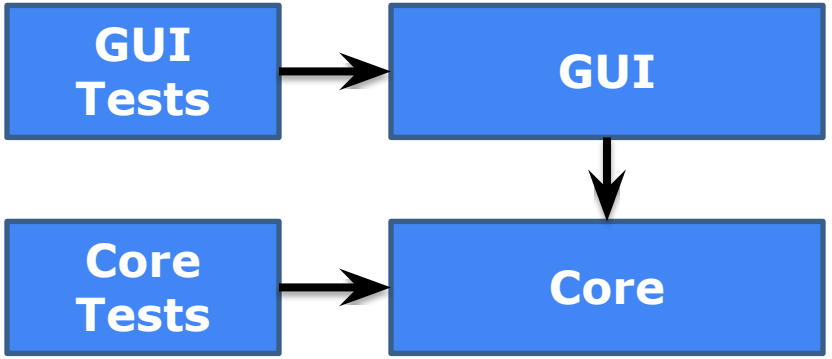
Recall the Observer



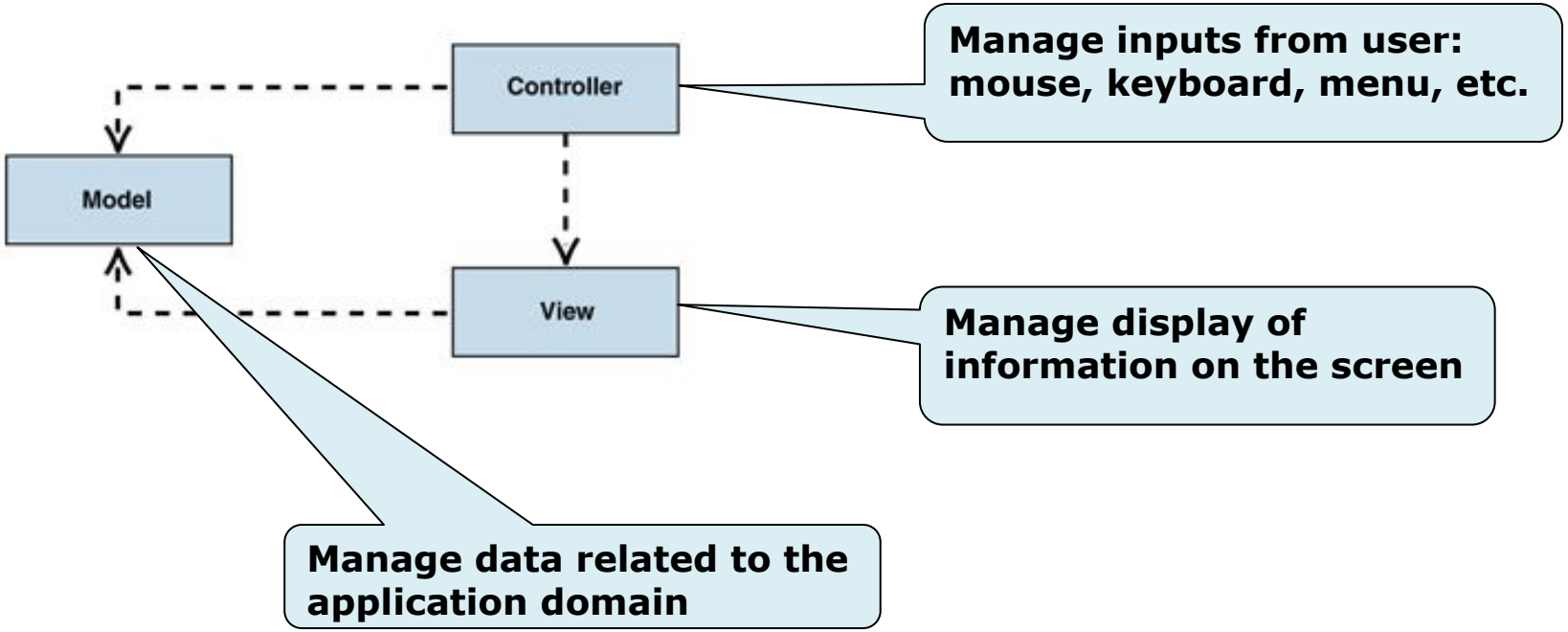
https://refactoring.guru/design-patterns/observer

Separating application core and GUI, a summary

- 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)

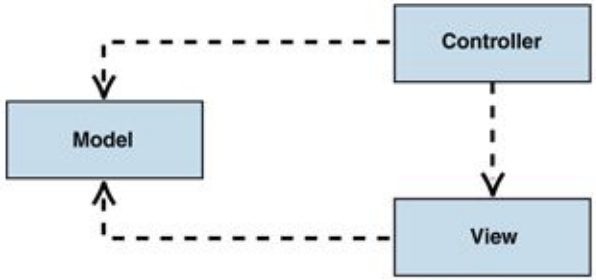


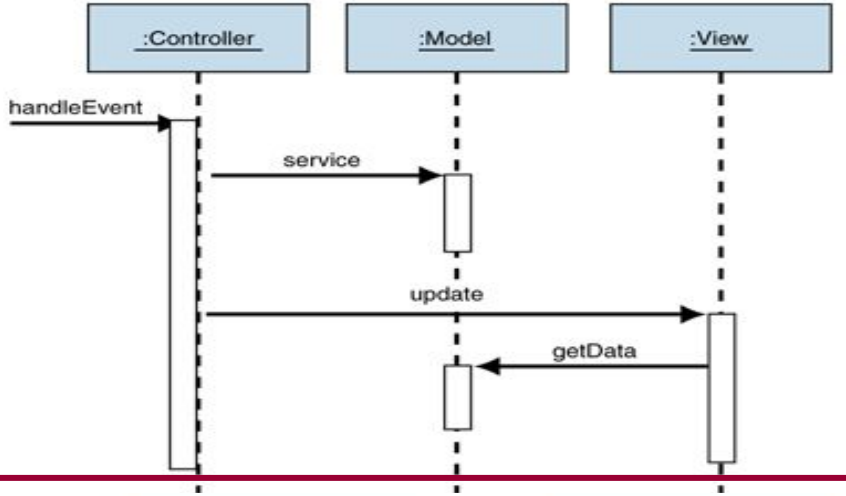
An architectural pattern: Model-View-Controller (MVC)



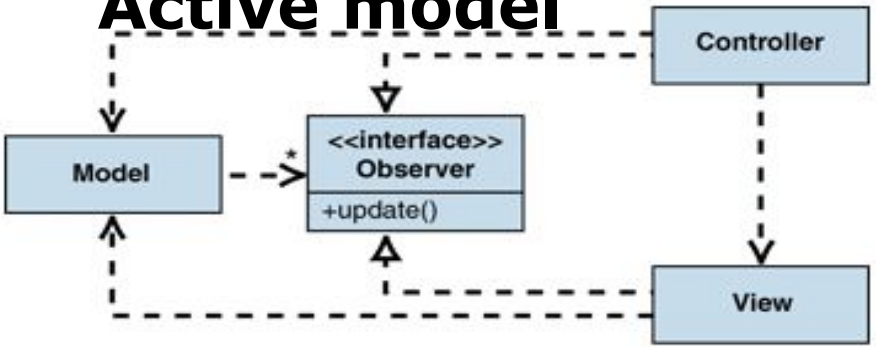
Model-View-Controller (MVC)

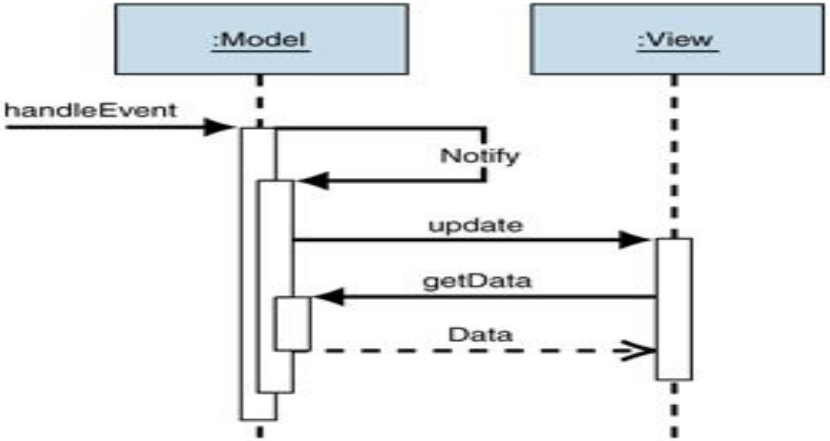
Passive model





Active model





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

Asynchrony

Asynchrony

- We use an asynchronous method call:
  - normally, when we need to do work away from the current application;
  - and we don't want to wait and block our application awaiting the response

Asynchrony

Usually, managing asynchronous events involves concurrency

- Do something while we wait
- Multiple events can overlap
- We will focus on constructs for handling both

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

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

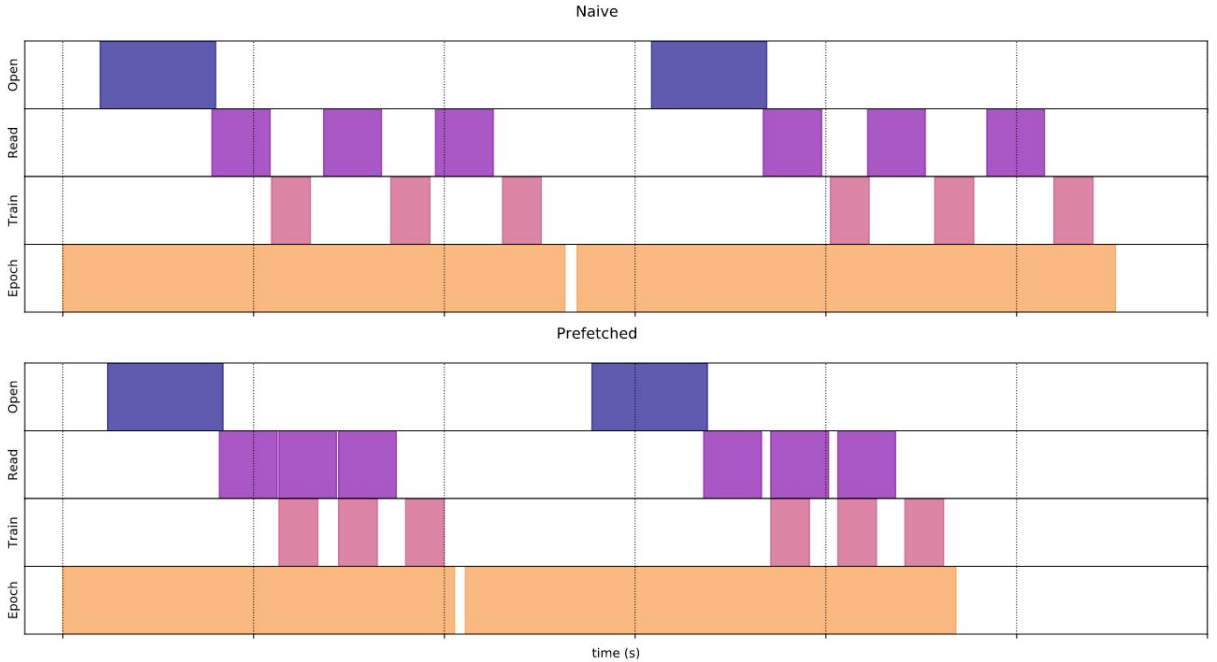
Can save you a lot of time

- 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: 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

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

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