Responsiveness

Human perception

Handling long tasks with worker threads

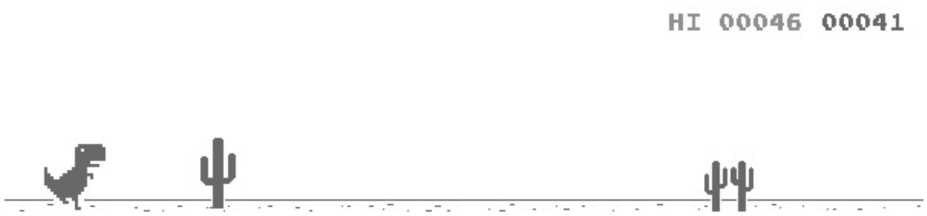
Responsive User Interfaces

- What is a responsive UI?
  - adapts to different window sizes and/or devices
    (we talked about responsive layouts previously)
  - delivers feedback in a timely manner
    (we talked about responsive layouts previously)
- We can make feedback responsive in two ways:
  1. loading data efficiently so it's available quickly
  2. designing for human perception of time

Human Perception of Time

Elevator 1

Elevator 2

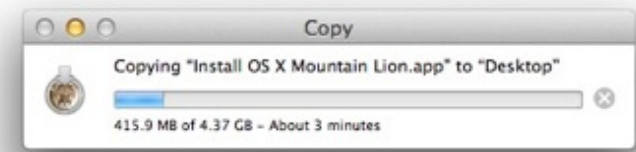


What factors affect responsiveness?

- User Expectations
  – how quickly a system “should” react, or complete some task
- Application and Interface Design
  – the interface keeps up with user actions
  – the interface informs the user about application status
  – the interface doesn’t make users wait unexpectedly
- **Responsiveness is the most important factor in determining user satisfaction,** more so than ease of learning, or ease of use
- **Responsiveness is not (necessarily) System Performance**

Slow Performance, but Responsive

- providing feedback to confirm user actions
  (e.g., let them know that their input was received)
- provide feedback about what is happening
  (e.g., indication of how long an operations will take).
- allow users to perform other tasks while waiting
- anticipate users' most common requests.
  (e.g. pre-fetch data below current scroll view)
- perform housekeeping and low-priority tasks in the background



Fast Performance, but not Responsive?

Perceived Time

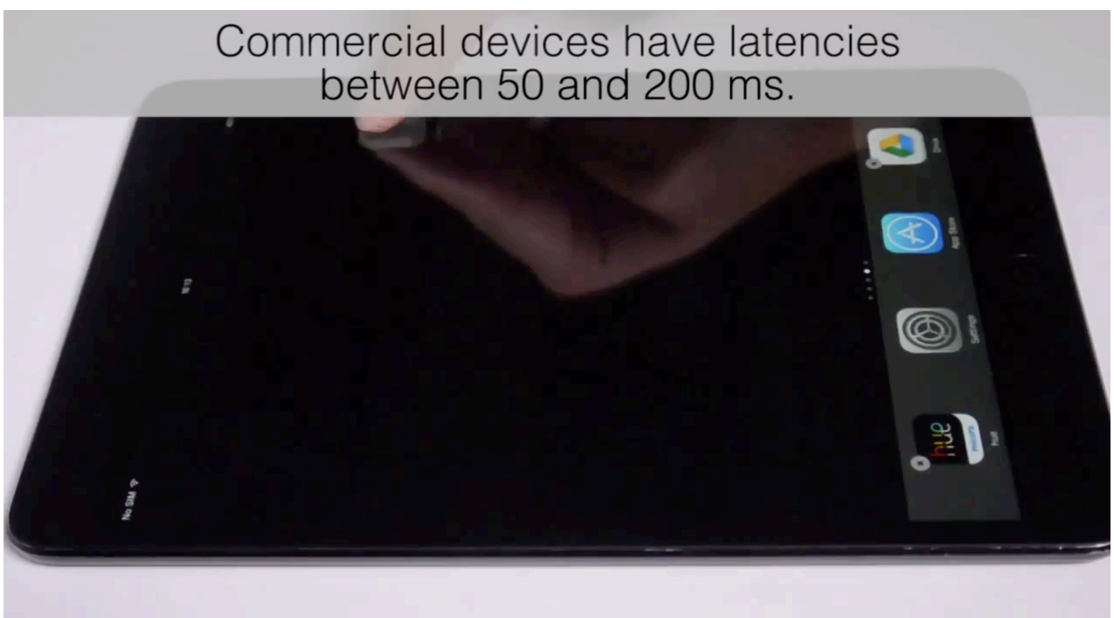
- Knowing the duration of perceptual and cognitive processes can inform the design of interactive systems that feel responsive

Minimal time to detect a gap of silence in sound	4 ms
Minimal time to be affected by a visual stimulus	**10 ms**
Time that vision is supressed during a saccade	100 ms
Maximum interval between cause-effect events	**140 ms**
Time to comprehend a printed word	150 ms
Visual-motor reaction time to inspected events	**1 s**
Time to prepare for conscious cognition task	**10 s**
Duration of unbroken attention to a single task	6 s to 30 s

(times approximate)

Example Design Implications

- **Minimal time to be affected by a visual stimulus**
  → continuous input latency should be less than 10ms
- **Maximum interval between cause-effect events**
  → if UI feedback takes longer than 140ms to appear, the perception of ”cause and effect” is broken

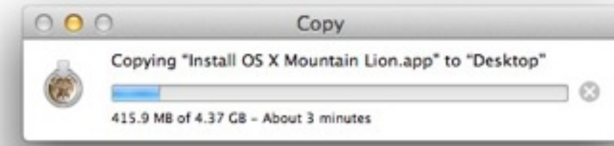


User Perception of Latency & Latency Improvements in Direct and Indirect Touch
– https://youtu.be/1dKlMZrM_sw

Example Design Implications

- **Visual-motor reaction time for unexpected events:**
  → Display busy/progress indicators for operations more than 1s

Busy Indicator

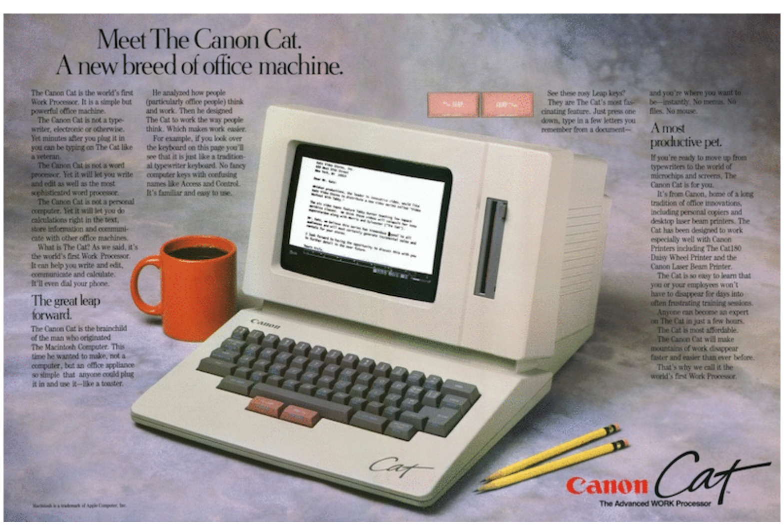


Progress Bar

→ Present a "fake" inactive version of an object while the real one loads in less than 1s (see next page for extreme version of this)

Design Implications

- **Time to prepare for conscious cognition task**
  → Display a fake version of an application interface, or image of document on last save, while the real one loads in less than 10s



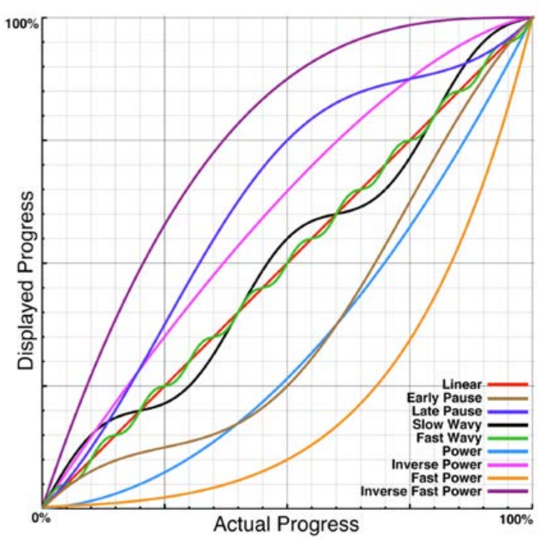
Progress Indicator Design Best Practices

- Show work remaining, not work completed
- Show total progress when multiple steps, not only step progress.
- Display finished state (e.g. 100%) very briefly at the end
- Show smooth progress, not erratic bursts
- Use human precision, not computer precision
  (Bad: “243.5 seconds remaining”, Good: “about 4 minutes”)

(McInerney and Li, 2002)

Responsiveness by Tweaking Progress Bars

- Change the mapping from *actual progress* to *displayed progress*



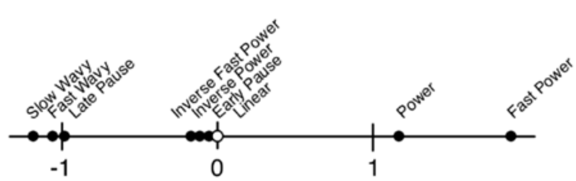


Figure 4: Number line showing relative distances from linear, which is centered at 0. Values generated from logistic regression model.

Harrison, C., Amento, B., Kuznetsov, S., and Bell, R. 2007. Rethinking the progress bar. UIST '07 http://www.chrisharrison.net/index.php/Research/ProgressBars

Harrison, C., Yeo, Z., and Hudson, S. E. 2010. Faster Progress Bars: Manipulating Perceived Duration with Visual Augmentations. CHI 2010

– https://www.youtube.com/watch?v=CDnN3wLY3OE

Responsiveness by Progressive Loading

- Provide user with some data while loading rest of data
- *Examples*
  - word processor shows first page as soon as document opens
  - search function displays items as soon as it finds them
  - webpage displays low resolution images, then higher resolution

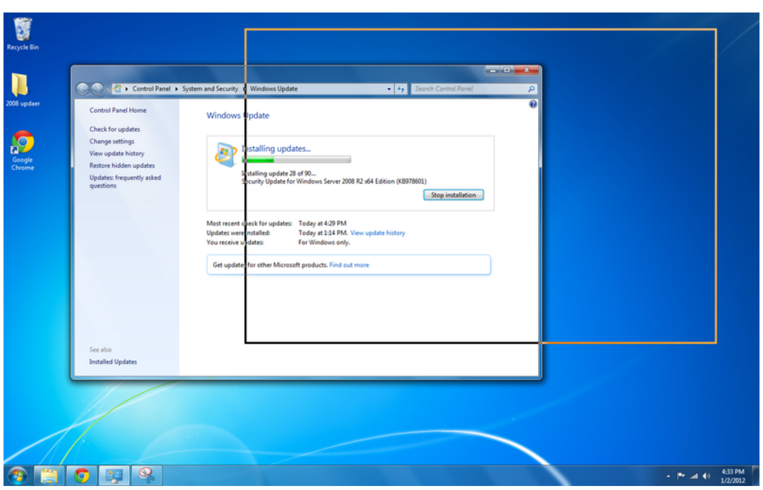
Responsiveness by Predicting Next Operation

- Use periods of low load to pre-compute responses to high probability requests. Speeds up subsequent responses
- *Examples*
  - text search function looks for next occurrence of the target word while user looks at the current
  - web browser pre-downloads linked pages ("pre-fetch")

Responsiveness by Graceful Degradation of Feedback

Simplify feedback for high-computation tasks

- *Examples*
  - base window system updates window after drag
  - graphics editor only draws object outlines during manipulation
  - CAD package reduces render quality when panning or zooming



Responsiveness by Chunking Processing

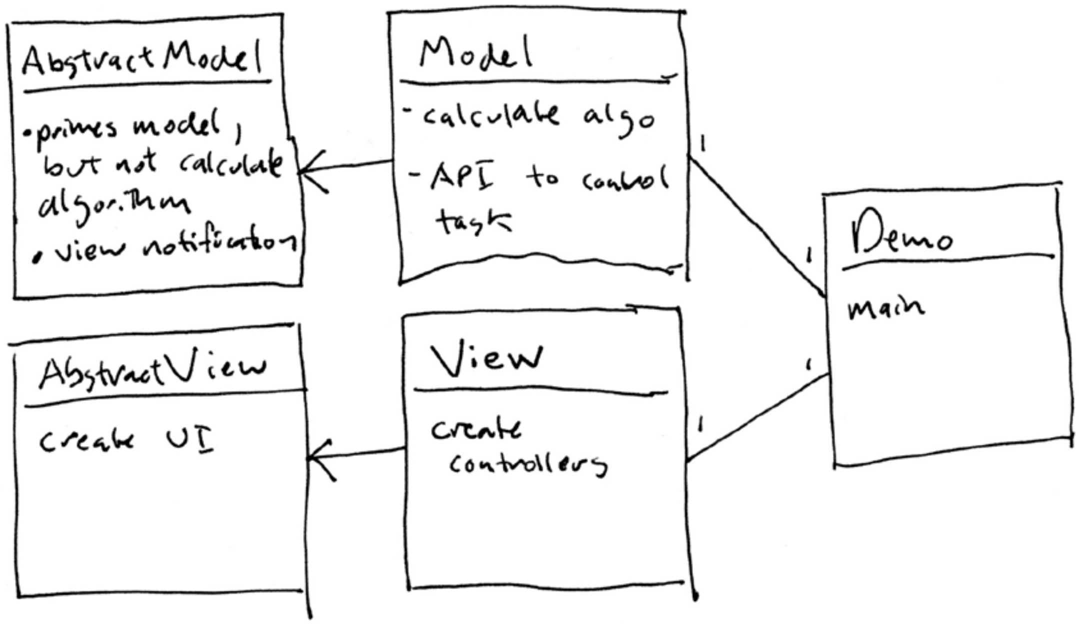
- Avoid doing frequent processing during user interaction
- *Example*
  - validate after pressing ENTER, not character by character
  - don't send data to server until after direct manipulation action

Bad Chunking Example
Car navigation system prompts the user to enter the City, then Street, then Address, and validates *every* keystroke.

Handling Long Tasks in a UI

- Goals
  - keep UI responsive
  - provide progress feedback
  - allow long task to be paused or canceled
- (even if it takes a bit longer to complete the task)

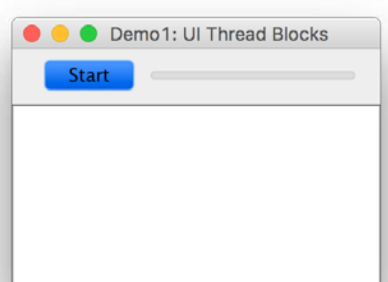
Demo MVC Architecture



Demo1.java (what not to do)

```
protected void registerControllers() {
   // Handle presses of the start button
   this.startStopButton.addActionListener(new
   ActionListener() {
          public void actionPerformed(ActionEvent e) {
               model.calculatePrimes();
          }
       });
}
```

Find primes in [1, 250000]
Takes ~10 seconds to complete



UI Execution Thread

- Island and Ocean Analogy (blackboard)

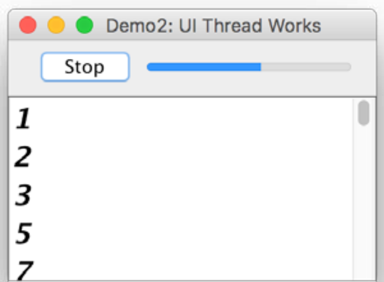
Two Strategies for Long Tasks

- **Strategy A: Run in UI Thread (by breaking into subtasks)**
  – Periodically execute subtasks between handling UI events
- **Strategy B: Run in different thread (worker thread)**
  – Use thread-safe API to communicate between worker and UI
- Both strategies let UI control task and let task send feedback to UI

```
void run()
void cancel()
boolean isRunning()
boolean isDone()
boolean wasCancelled()
int progress()
```

Strategy A: Run in UI Thread (Demo2.java)

- Task object keeps track of current task progress
- Subtasks periodically called on UI thread
  - uses `SwingUtilities.invokeLater()`
  - (could also use `javax.swing.Timer`)
- Every time object told to "run" for a bit, it checks current progress, executes subtask, updates progress, cancels if asked, …




```
class Model2 extends AbstractModel {

   private boolean cancelled = false;
   private boolean running = false;
   private int current = 0;        // progress so far

   public Model2(int min, int max) {  super(min, max);   }

   public void calculatePrimes() {
      this.running = true;
      SwingUtilities.invokeLater(new Runnable() {
         public void run() {
            // calculate primes for 100 ms
            calculateSomePrimes(100);
            if (!cancelled && current <= max) {
               calculatePrimes();
            }
         }
      });
   }
```

```java
private void calculateSomePrimes(long duration) {
   long start = System.currentTimeMillis();
   while (true) {
      if (this.current > this.max) {
         this.running = false;
         updateAllViews();
         return;
      } else if (System.currentTimeMillis() - start >=
                                                duration) {
         updateAllViews();
         return;

      } else if (isPrime(this.current)) {
         this.addPrime(current);
      }
      current += 1;
   }
}
```


Strategy A

- Advantages:
  - Can more naturally handle "pausing" (stopping/restarting) task because it maintains information on progress of overall task
  - Can be run in Swing event thread or separate thread
  - Useful in single-threaded platforms (e.g., mobile)
- Disadvantages:
  - Tricky to predict length of time for subtasks
  - Not all tasks can easily break down into subtasks (e.g., Blocking I/O)
- These are some big disadvantages, it's better to use threads (Strategy B) when possible

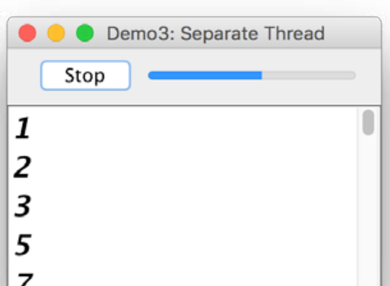
Threads and Multi-Threading

- **Thread:** the smallest "stream" of execution in a program
- **Multi-threading:** manage multiple concurrent threads with shared resources, but executing different instructions
- Threads are a way to divide computation, reduce blocking.
- Concurrency has risks: what if two threads update a variable?

- Typically three types of threads in a UI application:
  - one **main application** thread
  - one **UI Thread** (Java calls it Event Dispatch Thread (EDT)
  - 0 or more **worker threads** (also called "background threads")

Strategy B: Run in separate thread (Demo3.java)

- Long method runs in a separate thread
  - Typically implemented via Runnable object
- Method regularly checks if task should be cancelled and reports back to UI about progress (by updating views)



```java
class Model3 extends AbstractModel {
   ...

   public void calculatePrimes() {

      new Thread() {

         public void run() {
            ...
         }

         private void updateUI() {
            ...
         }

      }.start();
```



```java
public void run() {
   running = true;
   long start = System.currentTimeMillis();

   while (true) {
     if (cancelled || current > max) {
        running = false;
        updateUI();
        return;
      } else if (isPrime(current)) {
        addPrime(current);
      }
      current += 1;

      if (System.currentTimeMillis() - start >= 100) {
        updateUI();
        start = System.currentTimeMillis();
     }
   }
}
```

```java
// the synchronized keyword is needed to share Vector
across two threads
protected synchronized void addPrime(int i) {
   super.addPrime(i);
}

public synchronized Vector<Integer> getPrimes() {
   return super.getPrimes();
}
```


`synchronized` keyword

- Java's uses the **monitor** abstraction for concurrency
  - Conceptually higher level than *semaphores* and *mutexs*
  - Goals is to enforce exclusive access to critical sections
- synchronized methods can only be access by one thread a time
  - e.g. why synchronize methods the set or return a Vector? (addPrime and getPrimes in Demo3)

```java
private void updateUI() {

  // updateUI is called from worker thread, not UI thread
  // must give the update back to the UI thread to run
  SwingUtilities.invokeLater(new Runnable() {
     public void run() {
        updateAllViews();
     }
   });

}
```


Strategy B

- Advantages:
  - Conceptually, easiest to implement
  - Takes advantage of multi-core architectures
- Disadvantages:
  - Need to be careful about inter-thread communication
  - All the usual threading caveats: race conditions, deadlocks, ...

Thread-Safety

- Most UI toolkits (like Swing) are not **thread safe**
- Can't call toolkit methods or access widgets from other threads
- Invoke code to run on the UI thread:
  – e.g. `SwingUtilities.invokeLater`
- Handle concurrency by protecting critical sections of code
  – e.g. Java `synchronized` keyword
- http://docs.oracle.com/javase/tutorial/uiswing/concurrency/index.html

You may see this ...

- Remember there's a **main thread** and a **ui thread**
- For this reason, you may see something like this in Swing code:

```
public static void main(String[] args) {
  SwingUtilities.invokeLater(new Runnable() {
    public void run() {
      new createAndShowGUI();
    }
  });
}
```

- https://bitguru.wordpress.com/2007/03/21/will-the-real-swing-single-threading-rule-please-stand-up/

Why are GUI toolkits single-threaded?

- People are "single threaded": assumption is that interaction processing is single threaded too
- Two sets of abstractions flowing in opposite directions:
  - User-initiated threads travel "down" to the hardware to run (e.g. start a thread to find a primes)
  - Events travel from hardware up to higher-level abstractions (e.g. button-click to cancel finding primes)
- Any locking protocol for these two abstractions will conflict
- A long history of smart people trying to build thread-safe toolkits
- https://community.oracle.com/blogs/kgh/2004/10/19/multithreaded-toolkits-failed-dream

Toolkit Worker Thread Classes

- Swing has a worker thread class: SwingWorker
- Good introductory tutorial:
  - https://docs.oracle.com/javase/tutorial/uiswing/concurrency/worker.html
- Android has something similar, AsyncTask
  - http://developer.android.com/reference/android/os/AsyncTask.html

Long Tasks and MVC

- Long tasks start to break clean separation of MVC
- Model's methods need to be designed to allow user to stop them, to maintain interactivity
  - Needed to service event queue
  - Needed to allow user to stop method
- May need methods to inquire about length of task completion
  - Not part of "model" per se, part of interaction
- Usability concerns are thus directly influencing design of model to accommodate user interaction (that's ok)