Lecture 14

Design Patterns and Principles: Part 1

From xkcd Webcomics: https://xkcd.com/974/
Outline

- Design in a Nutshell
- Abstraction and Encapsulation
- Class Cohesion and Coupling
- Wrapper Classes
“Design-Focused” Projects (1/2)

• Projects up to Fruit Ninja were considered “foundation-focused”

• Projects for the remainder of the semester are considered “design-focused”
  o given only an assignment specification (and hints), you will design programs from scratch

• On early projects, design was 25% of code grade; now 30-35%
  o for DoodleJump and Tetris partnered projects, 20% of grade goes to individual check-in, where you will describe your design and code to TAs
“Design-Focused” Projects (2/2)

• You should put much more effort (at least 2-3 hours) into understanding assignment specifications and planning design **before coding** on design-focused projects
  
  ○ containment/association and interface/inheritance diagrams are crucial!

• Starting to code with a poor design can lead to hours wasted trying to design and code on the fly
Design Grading

• Your Cartoon design grade will be based on the design guidelines in the handout and that we’ve already discussed throughout this semester
  o will NOT be graded on specifics of this lecture

• The remaining projects’ design WILL be graded with this week’s design patterns + principles lectures in mind
  o refer to this lecture when designing your DoodleJump with your partner!
Outline

- Design in a Nutshell
- Abstraction and Encapsulation
- Class Cohesion and Coupling
- Wrapper Classes
Context Beyond CS15

• Imagine you’re working for a company with a bunch of software engineers that write the code for a popular app
• The app needs to work properly now, and in the future more engineers will need to add new features / change existing features
• Your job is to write code that:
  o works properly (functionality)
  o is easily readable (style)
  o another engineer can add to easily (design)
  o another engineer can modify easily (design)
• When writing real code, the **design** of your program is ultimately as important as its functionality
Design in a Nutshell (1/2)

- Up to now, focused on how to program
  - be appropriately lazy: re-use code and ideas
- Increasingly we learn about good design
- Some designs are better than others
  - “better” means, for example:
    - more efficient in space or time required (traditional criteria)
    - more robust, the “ilities” – usability, maintainability, extensibility, scalability…
- These are central concerns of Software Engineering
  - discussed in detail in CS32 (CSCI0320)
Design in a Nutshell (2/2)

● There are trade-offs to make everywhere
  o architect balances aesthetics, functionality, cost
  o mechanical engineer balances manufacturability, strength, maintainability, cost

● Need to defend your trade-offs
  o no perfect solution, no exact rules
  o up to now designs rather straight-forward, not concerned about performance because not dealing with larger collections of data
What Do We Cover in These Lectures?

• Walk through process of planning design for a mock CS15 project

• Emphasize design principles and design patterns, which will be directly relevant to projects (including DoodleJump!), along the way
Our Mock CS15 Project: Snake!

- Snake moves around board of squares at specified rate and continues moving in its last direction
- Player changes snake direction via key input, with goal of eating pellets to increase score
- Snake starts 3 squares in length, grows 1 square for each pellet eaten
- Snake can pivot right or left, but not 180°
- Gain score by eating pellets – different colors yield different scores
- Game ends when snake moves off screen or into itself
Software Development: A 5-Step Process
Software Development: A 5-Step Process

- **Analysis** has been done for us via assignment specification.

Design is where we’re focusing today!

- **Implementation** is when you code!

Testing, in CS15, typically means playing your game.

- **Maintenance** isn’t as applicable in CS15.
Outline

- Design in a Nutshell
- Abstraction and Encapsulation
- Class Cohesion and Coupling
- Wrapper Classes
Where do I start?!

• Assignment specifications can be daunting

• Start at the highest level: brainstorm how to separate components of program (delegation pattern!)
  o containment/association decisions
  o what classes should we write? how should they communicate with each other?
  o critical to consider where to divide abstractions
Recall: Delegation Leads to Abstraction

• Delegation results in a **levels of abstraction**, where each level deals with more specifics to complete an action

Please groom my dog!

Wash this dog with shampoo, then trim its hair and dry!

Fill the bath with warm water until it’s two-thirds full…

DogOwner → PetShop → DogGroomer

- Bath
- HairDryer
- Clippers
Abstractions (1/3)

- Each object represents an *abstraction*
  - a “black box”: hides details external users do not care about
  - allows you as the programmer to control programs’ complexity – only think about relevant features
Abstractions (2/3)

• CS15 support code and JavaFX are great examples of layers of abstraction

  andyBot.turnRight();
  this.pongBall.updatePosition();
  vbox.setAlignment(Pos.TOP_CENTER);

  don't need to worry about internals of how robot moves
  ball tracks its own speed and direction; we just tell it to move
  JavaFX handles pixel-level graphical representation
Abstractions (3/3)

- CS15 support code itself results in a levels of abstraction
  - each layer becomes more specific

```
PongGame
  this.pongBall.updatePosition();

CS15Ball
  circle.setCenterY(circle.getCenterY() + verticalChange);

JavaFX Circle
  JavaFX internals to manipulate specific pixels
```

This shows the “behind the scenes” of the CS15 support code!
Class Communication (1/2)

• Lack of clean abstractions leads to messy communication between classes

• Example: Game class contains composite shape class that moves across screen
  o think of each communication between Game and CompositeShape internals as one arrow connecting them
  o bad design because game uses composite shape internals to move it!

```java
shape.getHead().setX(20 + shape.getHead().getX());
shape.getBody().setX(20 + shape.getBody().getX());```

Game

CompositeShape
Class Communication (2/2)

• Clean abstractions leads to clear communication between classes

• Delegate details of moving individual shapes to CompositeShape to simplify communication
  ○ Game doesn’t need to know the details of moving shapes!

```java
shape.moveRight();

public void moveRight() {
    this.head.setX(20 + this.head.getX());
    this.body.setX(20 + this.head.getX());
}
```
Recall: Encapsulation Design Pattern

- Why `private` instance variables?
- **Encapsulation** for safety… your properties are your private business
- Allows for **abstractions** so classes don’t need to worry about the inner workings of contained classes
  - but we can allow selective access via setters and getters, only as necessary!

DogOwner → PetShop → DogGroomer

- Bath
- HairDryer
- Clippers
Encapsulation (1/2)

• If Game manages the movement of individual shapes, we must allow access to CompositeShape’s private components head and body
  □ via `getHead()` and `getBody()`

• With access to those shapes, Game could also write code like
  `this.getHead().setFill(Color.RED);`
  □ but what if we don’t want them to be allowed to change color?!
Encapsulation (2/2)

- Once we abstract away the details of moving shapes, no more need for `getHead()` and `getBody()`!

- `CompositeShape` can keep its instance variable privacy

- **Key Point:** Use getters/setters only as necessary to maximize encapsulation safety
Outline

- Design in a Nutshell
- Abstraction and Encapsulation
- Class Cohesion and Coupling
- Wrapper Classes
Review: Composition Pattern (1/2)

- You’ve used composition from the beginning

- Model objects that are composed of other objects or are associated with peer objects

- Factors similar code out into a different class accessed via references in instance or local variables. These references provide access protection through encapsulation
  - components, typically initialized in the constructor, e.g., engine
  - associations to peer objects, e.g., driver (via parameter passing)
Review: Composition Pattern (2/2)

- Compose one object out of other, more specialized objects
  - factor out similar code into a separate class
  - instantiate an instance of this class if you need that functionality
  - allows specialists to design components that you can build on
    - black boxes that expose only limited functionality
    - this is a form of delegation – don’t rewrite code that specialists can do for you

- A Car class would use instances of these classes to model a Car’s capabilities
  - would contain one or more instances of an Engine class, a Brake class, etc.
  - Car can delegate startUp() to the Engine

- How can we determine good delegation and composition decisions?
High Cohesion and Loose Coupling (1/3)

• Cohesion refers to how well-defined the purpose of a single class is

• Each class should have **high cohesion** with itself
  
  o **Single Responsibility Principle** – Each class has a single, well-defined purpose

• Strong **separation of concerns** reduces mental juggling; when coding in one class, only need to think about limited pieces of functionality

• You should be able to succinctly describe the purpose of each class in class header comments
## High Cohesion or Low Cohesion?

**High Cohesion**

- In a program representing the life of a student, there is one **CS15** class for the student to track their CS15 assignments.
- In Cartoon, one class that represents a **Cloud** with 5 circles and moves each of the circles across the pane.

**Low Cohesion**

- In a program representing the life of a student, there is one **Life** class that handles Fall classes, social life, and extracurriculars.
- In Cartoon, **PaneOrganizer** handles setting up the overall structure of panes, subpanes and shapes, and handles changing the color of each shape on key presses.
High Cohesion and **Loose Coupling** (2/3)

- Coupling refers to how interdependent two classes are
- Each class should have **loose coupling** with other classes
  - use **abstractions** to keep clear relationships between classes
- Limit dependencies between classes
  - should be able to modify internals of one class without worrying about the impact on other classes
Coupling Example (1/3)

• Back to shape movement! Let’s say we have our app to make a planet move via `Planet` class
  ○ to start, planet is just represented by a `Circle`

```java
public class Planet {
    private Circle circle;
    public Planet() {
        this.circle = new Circle(100);
    }
    public Circle getCircle() { return this.circle; }
}
```

// in Cartoon class
Planet venus = new Planet();
Timeline timeline = new Timeline(Duration.seconds(1), (ActionEvent e) ->
    venus.getCircle().setX(venus.getCircle().getX() + 10);
Coupling Example (2/3)

• Now we decide to use a composite shape with 4 rings around the planet
• First, we add the rings to Planet class
• Now every time we add a shape, we need to move it in Cartoon
• This is tight coupling, i.e., Cartoon is too involved with the details of moving Planet

// in Cartoon class
Planet venus = new Planet();
Timeline timeline = new Timeline(Duration.seconds(1), (ActionEvent e) -> {
    venus.getCircle().setX(venus.getCircle().getX() + 10);
    venus.getRing1().setX(venus.getRing1().getX() + 10);
    venus.getRing2().setX(venus.getRing2().getX() + 10);
    // etc.
});
Coupling Example (3/3)

- Alternatively, we could just have one `move` method in `Planet`
- `Planet` could have 1 shape or 18 shapes, and `Cartoon` doesn’t need to change!
- This is **loose coupling**

```java
public class Planet {
   // constructor and instance variables elided
   public void move() {
      // This method could move one shape or a bunch of shapes, but
      // Cartoon doesn’t need to know about the details!
   }
}
```

// in Cartoon class
Planet venus = new Planet();
Timeline timeline = new Timeline(Duration.seconds(1),
                                 (ActionEvent e) -> venus.move());
High Cohesion and Loose Coupling (3/3)

• Key Point: In short, each class should have an independent, well-defined purpose (**high cohesion**), and communication relationships between classes should be simple and well-defined (**loose coupling**).
A **SurvivorFan** class is using an instance of the **Remote** class to watch a season of Survivor. Which code in **SurvivorFan** would indicate that the **Remote** class is written with proper encapsulation, abstractions, and loose coupling?

A. `remote.getButtons().getPlayButton().press();`

B. `remote.pressPlay();`

C. `remote.pressButton("Play");
remote.triggerInternalRedWire();
remote.releaseButton("Play");`
Back to Snake Brainstorming (1/3)

• Start at the highest level: brainstorm how to separate components of program
  o keeping in mind aim of **high cohesion** and **loose coupling**

  - **PaneOrganizer** – Organize high-level graphical elements of game
  - **Game** – Handle high-level logic of game through timeline and key input
  - Data structure to represent board… 2D array of squares?
  - Data structure to represent snake… **ArrayList** because the length changes?

*We succinctly described purpose of these component – indicator of high cohesion!*
C/A Diagram Draft

App

PaneOrganizer

SnakeGame

Square
Back to Snake Brainstorming (2/3)

• Let’s think more about what’s going on in the Game class
• What should happen at each tick of the Timeline?
• Let’s write pseudocode:
  o move the snake into the next tile
  o if the snake went off the board or ran into itself:
    ▪ game over
  o if there is pellet on the tile the snake moved into:
    ▪ eat the pellet
    ▪ add to score
    ▪ generate a new pellet
    ▪ increase snake length by one square
Back to Snake Brainstorming (3/3)

• We realize that each board square needs some extra information
  o is the snake on the square?
  o is there a pellet on the square?
• With more complexity, let’s consider delegating to a class `BoardSquare` rather than making the Game track all of that
  o instead of a board of “simple squares” (javafx `Rectangles`), we need “smart squares” (our own `BoardSquare` class)
  o then we can model this extra information as `properties` (instance variables) of the square!
Designing the **BoardSquare** (1/3)

• Since each **BoardSquare** represents one graphical square, should we have **BoardSquare** inherit from a JavaFX **Rectangle**?

```java
public class BoardSquare extends Rectangle { …
```
Designing the **BoardSquare** (2/3)

- If `BoardSquare` extends `Rectangle`, the `BoardSquare` inherits all of `Rectangle`’s methods.

- That means the `BoardSquare` sets of public methods becomes the `Rectangle`’s set of public methods *plus* whatever specialized methods we write: is that a feature or a bug?

- In the context of Snake, we don’t want programmers to have access to all the `Rectangle` methods so they could change the position, size, rotation, etc. of the `BoardSquare`
Designing the **BoardSquare** (3/3)

- **Key point:** To achieve simple communication between classes *(loose coupling)*, the set of public methods a class or interface exposes should be as *simple* and *clear* as possible

- Remember **encapsulation**… keep private parts your own business

- Let’s only allow users of the **BoardSquare** to access the limited parts we *need* to make public

- In this case, most of **Rectangle** methods shouldn’t be accessible – how can we make them private?
Outline

- Design in a Nutshell
- Abstraction and Encapsulation
- Class Cohesion and Coupling
- Wrapper Classes
Wrapper Classes

• A wrapper is code that encapsulates (or “wraps” around) another software component as a layer of abstraction

• In Java specifically, we create wrapper classes that add a layer of abstraction to another Java class

• Instead of inheriting from a class, contain an instance of that class as a component (instance variable)
BoardSquare Wrapper Class (1/2)

- **BoardSquare** wraps an instance of **Rectangle**

- Allows us to both restrict certain accesses inherited from **Rectangle** and add extra pieces of information
  - **boolean** to indicate is there a pellet on the square?
  - another **boolean** to indicate is there snake on the square?

```java
public class BoardSquare {
    private Rectangle square;
    private boolean hasPellet;
    private boolean hasSnake;

    public BoardSquare(Pane pane) {
        this.square = new Rectangle();
        pane.getChildren().add(this.square);
        this.hasPellet = false;
        this.hasSnake = false;
    }
}
```
BoardSquare Wrapper Class (2/2)

• A wrapper class exposes just the information that needs to be public and no more!
  ○ generally via setter and getter methods

• To show the snake moving across the board, one way is to change color of the square to Black
  ○ so we add a setter for Color

```java
public class BoardSquare {

    private Rectangle square;
    private boolean hasPellet;
    private boolean hasSnake;

    public BoardSquare(Pane pane) {
        this.square = new Rectangle();
        pane.getChildren().add(this.square);
        this.hasPellet = false;
        this.hasSnake = false;
    }

    public void setColor(Color color) {
        this.square.setColor(color);
    }
}
```
Keep Class Relationships Simple! (1/2)

• Is `setColor` the best we can do to abstract away the internals of the square?

• For our game, we want:
  o square to turn black when snake goes over it
  o square to return to original color when snake moves off it

• With `setColor`, programmer could make square any arbitrary color – that shouldn’t happen!

```java
public class BoardSquare {
    private Rectangle square;
    private boolean hasPellet;
    private boolean hasSnake;

    public BoardSquare(Pane pane) {
        this.square = new Rectangle();
        pane.getChildren().add(this.square);
        this.hasPellet = false;
        this.hasSnake = false;
    }

    public void setColor(Color color) {
        this.square.setColor(color);
    }
}
```
Keep Class Relationships Simple! (2/2)

• Instead, let’s have two separate methods
  o one method for the snake moving onto square
  o one method for snake leaving square
• Trade-off: this produces more code but makes relationship between classes simpler (looser coupling)
• Key Point: Strive for simpler class relationships – that may not always mean fewer methods!

```java
public class BoardSquare {
    // instance variable decl. elided
    public BoardSquare(Pane pane) {
        this.square = new Rectangle();
        pane.getChildren().add(this.square);
        this.hasFood = false;
        this.hasSnake = false;
    }
    public void enterSnake() {
        this.hasSnake = true;
        this.square.setColor(Color.BLACK);
    }
    public void leaveSnake() {
        this.hasSnake = false;
        this.square.setColor(Color.GREEN);
    }
}
```
Containment over Inheritance

• Wrapper classes are a good example of a generally agreed-upon design principle that containment is preferred to inheritance, unless the inheriting class should publicly expose all methods inherited (BoardSquare is-a Square vs. BoardSquare has-a Square)

• IntelliJ even has a button for “replace inheritance with delegation”
  o (but you should plan your design in advance so that you don’t have to use it)
TopHat Question

Which of the following is NOT true about wrapper classes?

A. The goal of a wrapper class is to make a class’s set of public methods as simple as possible
B. Wrapper classes are an example of using encapsulation
C. Wrapper classes add a layer of abstraction around some contained class
D. Wrapper classes use inheritance rather than composition
Representing the Snake (1/2)

• Let’s consider how to use `ArrayList` to represent the snake

• What should the `ArrayList` hold?
  
  o `BoardSquares` – hold whichever squares the snake is on top of
  
  o type will be `ArrayList<BoardSquare>"`
Representing the Snake (2/2)

- **ArrayList** could be an instance variable in Game class... or we could delegate it!
  - delegate for better **cohesion**
- Snake class will act as a **wrapper** class for the `ArrayList<BoardSquare>` and only expose a method to `move` and `changeDirection`
  - so much simpler than including all the `Rectangle` methods
- Game class doesn’t have direct access to **ArrayList** so it can’t mess with the contents of the list directly
Representing the Board

- We know our board will be 2D array `BoardSquare[][]`
- Once the board is created, we don’t want the contents to change
- Delegate to a `Board` class that acts as a wrapper of `BoardSquare[][]`?
  - definitely **high cohesion** since that class would only handle the board contents
  - no major benefit to delegating to `Board` as a wrapper since likely the only method would be a getter
  - could argue for or against having separate `Board` class
### Recap of Design Brainstorming So Far (1/2)

<table>
<thead>
<tr>
<th>Class</th>
<th>Purpose</th>
<th>Important Instance Variables</th>
<th>Important Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>App</td>
<td>Starts the application</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>PaneOrganizer</td>
<td>Organizes the high-level graphical organization of the program</td>
<td>BorderPane root</td>
<td>n/a</td>
</tr>
<tr>
<td>SnakeGame</td>
<td>Handles high-level logic of game via timeline and key input</td>
<td>Pane gamePane, Snake snake, Board board</td>
<td>updateGame (called on timeline), handleKeyInput (called on key press)</td>
</tr>
</tbody>
</table>
## Recap of Design Brainstorming So Far (2/2)

<table>
<thead>
<tr>
<th>Class</th>
<th>Purpose</th>
<th>Important Instance Variables</th>
<th>Important Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Snake</td>
<td>Represents snake moving around the board</td>
<td><code>ArrayList&lt;BoardSquare&gt; snakeSquares, Board myBoard</code></td>
<td><code>move, changeDirection</code></td>
</tr>
<tr>
<td>Board</td>
<td>Represents board of squares</td>
<td><code>BoardSquare[][] board</code></td>
<td><code>getTileAt</code></td>
</tr>
<tr>
<td>BoardSquare</td>
<td>Represents one square on the board</td>
<td><code>Rectangle boardSquare</code></td>
<td><code>enterSnake, leaveSnake</code></td>
</tr>
</tbody>
</table>
Containment/Association Diagram

- **App**
- **PaneOrganizer**
- **SnakeGame**
- **Board**
- **Snake**
- **BoardSquare**

- BoardSquare via 2D array
- Snake via ArrayList
Class Diagram

App

PaneOrganizer
- root : BorderPane

SnakeGame
- updateGame() : void
- handleKeyInput(KeyEvent e) : void

Board
- board : BoardSquare[][]
+ getTileAt(int row, int col)

Snake
- snakeSquares : ArrayList<BoardSquare>
+ move() : BoardSquare
+ changeDirection(Direction dir) : void

BoardSquare
- square : Rectangle
+ enterSnake()
+ leaveSnake()
Announcements (1/2)

- **Snake code on GitHub** – check it out to see contrasting design decisions, and example of large program implementation
  - will continue during Thursday’s lecture

- **ArrayList + Loops Section this week!**
  - be sure to complete [mini-assignment](#) and send to section TAs prior to section

- **DoodleJump deadlines**
  - early handin: Monday 11/1
  - on-time handin: Wednesday 11/3
  - late handin: Friday 11/5
  - do not underestimate this assignment! start early!
Announcements (2/2)

• Late days clarifications
  o 3 late \textit{days} (not late passes!), can apply up to 2 on a project
  o on partner projects, your team can use as many (up to 2) as the maximum left between you both

• Extra HTA hours this week – come talk to us if you have concerns about partner project!
  o today 10/26 @ 5pm in Salomon DECI
  o Thursday 10/28 @ 10am in Sayles 200
  o Friday 10/29 @ 3pm in Friedman 101

• Debugging hours – Still sign up individually, can attend on your own, or with your partner
DoodleJump: Questions to Get Started

• What classes should you represent in DoodleJump? What should their containment/association relationships be?

• How can you leverage “wrapper classes” to wrap some JavaFX elements you use to represent components of the program?

• How can you model properties like game score and doodle velocity? Which classes are those properties of?

• What do the different platforms have in common, and how are they different? How can you leverage polymorphism to make it so that the game doesn’t need to know the actual type of each platform it moves?
Topics in Socially-Responsible Computing

Artificial Intelligence I
What is artificial intelligence? (approximately!)

**Artificial Intelligence**
the ability of a machine to perform “intelligent” tasks (predicting outcomes, classifying inputs, learning, planning, or perception).

**Machine Learning**
the ability of a machine to “learn” / gain takeaways from data using statistical/mathematical methods.

**Deep Learning**
a subset of ML based on a simplified model of the human brain (artificial neural networks).
How does ML work?

• Given input and output, ML does trial and error to find the in-between steps, elaborate pattern recognition
• “In-between steps” – neural networks
  • perceptron: hugely simplified notion of a neuron / how brain signals
  • perceptron takes several inputs and does weighted summation, “fires if sum is above a threshold
  • Evolution final project uses neural networks!
• Deep learning: learn from training data + apply to new data
• Reinforcement learning: dynamically learn by trial and error + adjust based on result
• Applications everywhere (pattern recognition!)
  • voice/image/facial recognition
  • autocomplete
  • content moderation (recall Facebook!)
  • “personalization” / content recommendation
  • autonomous vehicles (last Thursday!)
  • weather forecasting
  • translation / text generation / “writing” (e.g. GPT-3)
  • so many others!
• “AI 100 Report” released this year, led by Brown prof Michael Littman!
Applications

• Example: AlphaGo (2015)
  • 2015: DeepMind AI’s AlphaGo won first ever game against a Go professional 5-0
  • previous AI trained based on game strategies
  • only training was the rules of the game (very simple!), used reinforcement learning to simulate 4.9 million games of Go, brute force approach
  • pick moves that are expected to have better outcomes!
  • AlphaZero, successor, mastered chess, Go, and shogi (diff game) at superhuman level within 24 hours

• GPT-3, new language model to produce human-like text
  • GitHub Co-Pilot: can autocomplete code
  • can write paragraphs of text based on prompt (incl news articles, easy to produce fake news + w/ other tech news broadcasts w/ voice + synthetic anchors)
Positive use case (diagnosis)

- “Six studies demonstrated that AI can aid in diagnosis [of breast cancer], with up to 69% reduction in false positives and an increase in sensitivity ranging from 84 to 91%.” (Oncology, scientific journal)

- “In most applications, machine learning shows better performance than manual segmentation of the brain tumors from MRI images as it is a difficult and time-consuming task.” (Nature, scientific journal)
Positive use case (improving accessibility)

• Sound-based user interfaces
  • blind/low vision people cannot often use standard GUIs, use voice recognition/generation
  • examples: Siri, Alexa, Dragon

• Auto-captioning for video
  • deaf/hard of hearing people (and more!) rely on closed captions to understand video
  • examples: Google Live Caption, Otter.ai (for Zoom and Google Meet!)

• Live image recognition (in development)
  • help blind/low-vision people access print documents, vision cues in surroundings
  • examples: Microsoft Seeing AI
Concerning use cases (1/2)

• Amazon tried to automate process of recommending / hiring workers in 2014
  • idea: give a model 100 resumes, it gives you the best 5 to interview
  • used dataset of ten years’ worth of resumes of people who currently worked there + trained on statistical model of 50,000 words
  • less importance to common engineering terms b/c they showed up in all resumes, subtle cues, preferred terms like “executed” and “captured"

• System consistently downgraded resumes from women
  • resumes w/ word “women” downgraded along with people who attended women’s colleges
  • “The vast majority of engineers hired by Amazon over ten years had been men, so the models they created, which were trained on the successful résumés of men, had learned to recommend men for future hiring.” — Kate Crawford, Atlas of AI
  • “The model was biased against women not just as a category but against commonly gendered forms of speech.”
Concerning use cases (2/2)

• “[ML] uses past data and projects it forward” — danah boyd ’00

• If training data is biased in any way, the workings of the neural network will be biased

• ML reflects human intelligence rather than necessarily superseding or working outside of it

• Predictive policing / “risk assessment”
  • COMPAS recidivism algorithm
  • Black defendants twice as likely as white defendants to be misclassified as higher risk
  • controlling for prior crimes, future recidivism, age, and gender, Black defendants 77% more likely to get higher risk scores

• Also
  • mortgage lending (“digital redlining”)
  • creditworthiness
  • so many others!
More reading that may be of interest!

- “Machine Bias: There’s software used across the country to predict future criminals. And it’s biased against Blacks” — Julia Angwin, Jeff Larson, Surya Mattu, Lauren Kirchner, ProPublica (2016)
- “Redlined by Algorithm” — Michelle Chen, Dissent (2019)
- “Google DeepMind: Ground-breaking AlphaGo masters the game of Go” — YouTube
- Atlas of AI – Kate Crawford
- “Amazon scraps secret AI recruiting tool that showed bias against women” — Jeffrey Dastin, Reuters (2018)
- “Deep Learning to Improve Breast Cancer Detection on Screening Mammography” — Nature
- “Deep Learning May Detect Breast Cancer Earlier than Radiologists” — Health IT Analytics
- “A Review of Applications of Machine Learning in Mammography and Future Challenges” — Oncology
- “A Review of Various Machine Learning Techniques for Brain Tumor Detection from MRI Images” — NIH
- “Microsoft’s incredible new app helps blind people see the world around them — take a look” — Insider
- “Microsoft 'Seeing AI': Imagining a New Use for Computer Vision” — Brien Possey, Redmond Mag (2021)