SWE 621

FALL 2020

DESIGN AS ABSTRACTION

IN CLASS EXERCISE

▸ What is an abstraction?

WHAT IS AN ABSTRACTION?

▸ The ability to interact with an idea while safely ignoring some of its details.

▸ A set of operations on shared state that make solving problems easier.

▸ Examples

   ▸ Data: String

   ▸ Collections: array, list, stack, queue, map, set, ...

   ▸ Big data: MapReduce, BigTable, Spanner

   ▸ AI: TensorFlow

   ▸ Web: HTTP request, HTTP response

   ▸ Business: Person, Party, Organization

LOGISTICS

▸ HW1 due today

▸ HW2 due in two weeks

ABSTRACTION AS MECHANISM FOR REUSE

▸ Abstractions serve as a mechanism for reuse of functionality

▸ Stakeholders in reuse

  ▸ Author: developer implementing the abstraction

  ▸ User: developer that is using the abstraction in their own code

▸ Often, a developer may be both an author and a user

▸ May have multiple authors, who may change over time

▸ For important abstractions, usually many more users than authors

CRAFTING ABSTRACTIONS

▸ Where do elements come from?

   ▸ Last time: from the domain model

▸ But... sometimes there are technical implementation considerations that lead to better ways of grouping functionality into elements

▸ Goal: choose elements that make solving the underlying problem easier

IN-CLASS ACTIVITY

▸ Write a function to reverse a List

▸ Available operations on elements in linked list

  ▸ Class ListElem

  ▸ {

      ▸ public ListElem getNext()

      ▸ public void setNext(ListElem e)

    ▸ }

IN CLASS ACTIVITY

▸ Write a function to reverse a list

▸ Available operations on a list

▸ class List {

  ▸ get(i)

  ▸ set(i)

  ▸ remove(i)

IN-CLASS ACTIVITY

▸ Write a function to reverse a List

▸ Available operations on elements in linked list

  ▸ getNext

  ▸ setNext

  ▸ getPrev

  ▸ setPrev

EXAMPLE: LIST

▸ State: an ordered set of elements

▸ Key operations

  ▸ add

  ▸ set

  ▸ get

  ▸ contains

  ▸ remove

  ▸ size

```java
List<Integer> l1 = new ArrayList<Integer>();
l1.add(0, 1);   // adds 1 at 0 index
l1.add(1, 2);   // adds 2 at 1 index
System.out.println(l1);  // [1, 2]
```

EXAMPLE: LIST

▸ User can be oblivious about how state is stored

  ▸ Could be linked list, could be array, could be stored locally, could be stored on another computer

▸ Supports a wide range of typical interactions with a list

▸ Abstraction author has wide range of implementation options

EXAMPLE: MAPREDUCE

▸ Organize computation into a map function that generates a new list from an old list and a reduce function that generates one (or a few) elements from a whole list

▸ Operations

  ▸ Map(k1,v1) → list(k2,v2)

  ▸ Reduce(k2, list (v2)) → list(v3)

https://storage.googleapis.com/pub-tools-public-publication-data/pdf/16cb30b4b92fd4989b8619a61752a2387c6dd474.pdf

EXAMPLE: MAPREDUCE

▸ Can distribute computation down to elements in the list to separate servers, which can work in parallel.

▸ Infrastructure

  ▸ marshals distributed servers

  ▸ runs tasks in parallel

  ▸ manages communication

  ▸ provides redundancy and fault tolerance

▸ Lets abstraction users focus on the computation to be done and let infrastructure worry about how to parallelize it

▸ Applicable for parallelizing a wide variety of typical computations

EXAMPLE: REACT COMPONENT

https://reactjs.org/

▸ State: properties (readonly, initialized by parents), state (changes over time)

▸ Operations

　▸ render

　▸ event listeners

　▸ set state

▸ Whenever state changes, render function is automatically called.

▸ Components organized into hierarchical trees. When data changes, generates new child components.

```
class Timer extends React.Component {
  constructor(props) {
    super(props);
    this.state = { seconds: 0 };
  }

  tick() {
    this.setState(state => ({
      seconds: state.seconds + 1
    }));
  }

  componentDidMount() {
    this.interval = setInterval(() =>
this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>
        Seconds: {this.state.seconds}
      </div>
    );
  }
}

ReactDOM.render(<Timer />, mountNode);
```

EXAMPLE: REACT COMPONENT

▸ React components do not need to worry about incrementally changing output in response to every event

  ▸ Would be complicated to figure out for every possible state change how to update output

▸ Instead, simply generate all new output whenever state no longer consistent with output

▸ Components focus on state and output for single element of interface

  ▸ Can be reused in many contexts because loosely coupled to parent and other ancestors

IN CLASS ACTIVITY

▸ Form a group of 2

▸ What's an abstraction you use frequently?

▸ What state does it have?

▸ What are the key operations?

▸ How does the abstraction simplify typical scenarios that occur?

BENEFITS OF GOOD ABSTRACTIONS

‣ Interoperability - can pass common data structures around

  ‣ Really important for library interop

‣ Can think about the problem without having to think about some low level details

  ‣ How is your data stored

  ‣ How computation is distributed to different servers in cluster

‣ Can predict behavior of operations, without reading implementation

  ‣ If common abstraction, that users are likely to be familiar with already

CHARACTERISTICS OF A GOOD ABSTRACTION

▸ Should do one thing and do it well

  ▸ If hard to name, that's a bad sign

▸ Implementation should not leak into abstraction

  ▸ If there's details that do not need to be exposed, do not

▸ Names matter

  ▸ Be self-explanatory, consistent, regular

CHALLENGES

▸ What operations to include? (a.ka., interface)

  ▸ Choices of operations has many consequences

▸ Not supporting necessary operations with state may make it impossible to use it in desired way, or lead to inefficient client code

▸ Supporting fewer operations may cause client code to be repetitive

▸ Operation choices may constrain design space of implementations

▸ If different users have slightly different needs, how do you balance conflicts?

IN CLASS ACTIVITY

▸ What's the most annoying abstraction you've ever used?

▸ What made it so hard to use?

HOW TO DESIGN A GOOD ABSTRACTION

Adapted from How to Design a Good API and Why it Matters, Joshua Bloch
https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/32713.pdf

GATHER REQUIREMENTS

▸ Often get proposed solutions instead

▸ Your job is to extract true requirements

  ▸ Should exist as scenarios where you abstraction will be used

  ▸ What does user want to accomplish in this scenario?

START WITH SHORT 1 PAGE DRAFT

▸ Focus on key ideas rather than completeness

▸ Bounce draft off as many people as possible

▸ What additional scenarios do they suggest?

▸ How well does your abstraction support these scenarios?

▸ How can it support these better?

RULE OF THREES

▸ Should try out abstraction with at least three scenarios

▸ Iterate design based on scenarios, ideally before publicly releasing

▸ How can you make these typical scenarios easier for users?

▸ How can you enable more efficient implementations?

SUMMARY

▸ Abstractions shape how you write code and think about a problem

▸ Design abstractions that cleanly capture typical operations on element at the right level of detail

▸ Good abstractions reduce boilerplate and let you focus on core problems.

▸ May require refactoring, as you have deeper insight into how to represent key ideas more clearly

▸ Important to keep abstractions consistent across team. Having similar but competing abstractions leads to confusion and conversion boilerplate.

IN CLASS ACTIVITY, STEP 1: BUILD ABSTRACTION

▸ Form groups of 2 or 3

▸ Build a abstraction(s) for a company org chart.

  ▸ Each employee has a 0 or 1 bosses and 0 to n subordinates

  ▸ Employee may direct one or more operating units, divisions, groups, or teams

  ▸ Operating units contain divisions

  ▸ Divisions contain groups

  ▸ Groups contain teams

▸ Deliverable: for each element you create, describe member elements in a class implementing this abstraction including

  ▸ State: what member variables does it contain

  ▸ Operations: what methods does it define and what is their signature

STEP 2: USE ABSTRACTION

▸ Switch abstractions with another group

▸ Using the other group's abstractions, sketch an algorithm to promote a division to an operating unit. Each group inside division remains a group.

DESIGN ACTIVITY: DISCUSSION

▸ What did you learn about the practice of design from this activity?