Program Design and Implementation

Program Design and Representation

- *program design* refers to identifying the classes and their methods
  - the projects have provided some examples of how complex programs can be made more manageable by breaking them up into chunks
    - data structures and related operations – `SolitaireDeck, Board`
    - other data types – `Block`, `Polyomino`, `Piece`
    - functionality – `Game`, `KeystreamGenerator`
- *representation* refers to deciding on what information is important to store and how to store it
  - we've seen different ways to store collections of values
    - concrete data structures (arrays, linked lists, binary trees)
    - arrangement / ordering of values (where the top of stack / front of queue goes, unsorted, sorted, binary search tree)
    - ADTs (List, Stack, Queue, PriorityQueue, Map, Set, …)

Program Design – Fundamentals of OOAD

- OOAD – object-oriented analysis and development

The idea of object-oriented programming is that the organization of the program should match how you think and talk about the problem.

- a program manipulates values that represent the ideas in the problem
  - classes reflect key concepts/things
    - often things which need some kind of representation (data storage) in the program, but classes can also exist only to group together related functionality
  - instance variables store information about those things
  - methods provide ways to access/use/manipulate the stored information

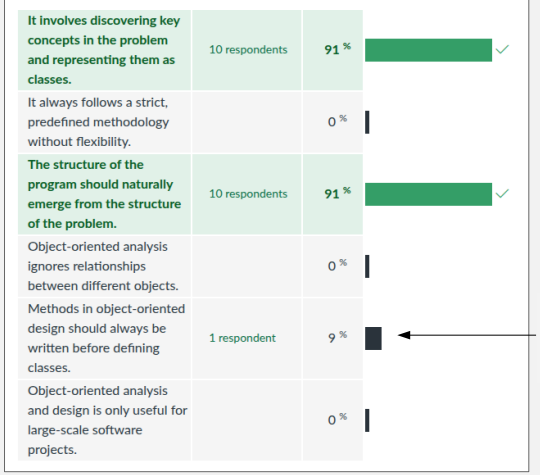
Program Design – Fundamentals of OOAD

Other goals of object-oriented design –

- modularity
  - because small, independent chunks are easier to understand (and reuse)
- encapsulation and information hiding
  - because it is easier to understand a chunk if you don't have to deal with all the details of how it does what it does (*information hiding*)
  - because isolating implementation decisions means you can change your mind about them – or support multiple alternatives – without changing the rest of the program (*encapsulation*)

→ want to group related values together into an object and protect the actual variables by providing only appropriate methods to manipulate those values

Which of the following statements about object-oriented analysis and design are true? Choose all that apply.



methods apply to objects

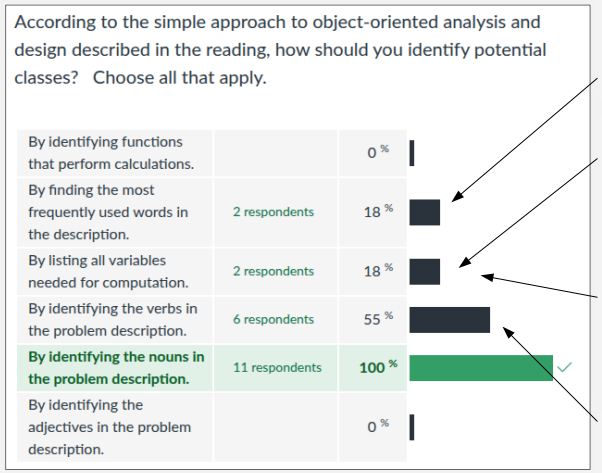
classes should be identified first because that gives methods a home

Textual Analysis

Textual analysis is a simple strategy that provides a starting point for identifying classes and methods.

- classes reflect key concepts/things, and nouns refer to things
  - identify nouns as potential classes
  - but not every noun – some may be synonyms or things that don't need to be represented in the program
- methods provide ways to access/use/manipulate the stored information, and verbs refer to processes
  - identify verbs as potential methods
  - when considering future reuse, also include operations that make sense for the concept even if not specifically needed for this application

According to the simple approach to object-oriented analysis and design described in the reading, how should you identify potential classes? Choose all that apply.



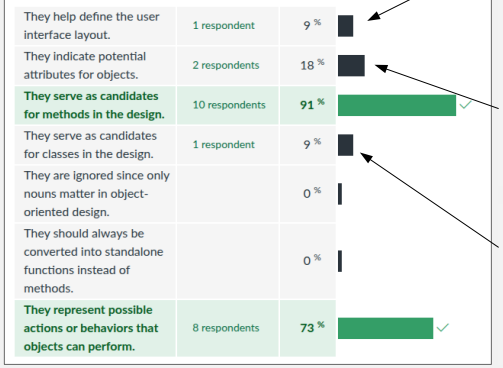
while the words for key concepts that will become classes are likely to be occur more frequently, frequency alone isn't the criteria for a potential class

identifying variables focuses on values, while classes are about types for those values

while thinking about values provides a prompt for thinking about the types of those values, identifying specific variables gets farther into specific algorithms and implementation details than we want at the design stage

verbs describe operations (methods), not concepts and things (classes)

According to the simple approach to object-oriented analysis and design described in the reading, what role(s) are played by the verbs in the problem description? Choose all that apply.



the user interface is what the user sees and interacts with when they use the program – while there may need to be elements to allow the user to do actions that are verbs in the problem description, OOAD is about the design of the program's code and not the appearance of the UI

attributes involve values – player's score, player's name – which are nouns rather than verbs (and will be instance variables rather than methods)

verbs translate into methods which are part of the class definition, but they don't correspond directly to classes themselves
e.g. "add the card to the hand" translates into an `add` method in the `Hand` class, not an `AddCard` class – "add card" isn't a kind of thing

Example

In a typical card game, each player gets a hand of cards. The deck is shuffled and cards are dealt one at a time from the deck and added to the players' hands. In some games, cards can be removed from a hand, and new cards can be added. The game is won or lost depending on the value (ace, 2, 3, ..., king) and suit (spades, diamonds, clubs, hearts) of the cards that a player receives.

- things
  - card
  - hand of cards
  - deck


- methods – access/manipulation of the stored info
  - card – get value, get suit
  - hand – add card (at the end), add card at a position, get card at a position, remove card at a position
  - deck – shuffle, deal card

https://freesvg.org/nine-of-spades-playing-card-vector-illustration
https://commons.wikimedia.org/wiki/File:Hand_of_cards.jpg
https://www.publicdomainpictures.net/en/view-image.php?image=11873&picture=deck-of-cards

Completing the Design

- identify instance variables – consider information storage
  - textual analysis – look for attributes associated with the concepts that become classes
    - e.g. player's score
  - consider *representation* – what values capture the concept of each class? what values are referenced or manipulated by the methods identified?
- complete the design
  - is all of the program's functionality accounted for?
    - e.g. main program
    - classes may also exist primarily to gather together related functionality
  - is everything the program needs to keep track of accounted for?
    - may be local variables in main or another class or additional instance variables in an already-identified class

Example

In a typical card game, each player gets a hand of cards. The deck is shuffled and cards are dealt one at a time from the deck and added to the players' hands. In some games, cards can be removed from a hand, and new cards can be added. The game is won or lost depending on the value (ace, 2, 3, ..., king) and suit (spades, diamonds, clubs, hearts) of the cards that a player receives.

- things
  - card
  - hand of cards
  - deck


- methods – access/manipulation of the stored info
  - card – get value, get suit
  - hand – add card (at the end), add card at a position, get card at a position, remove card at a position
  - deck – shuffle, deal card
- instance variables – relevant information about the things
  - card – value, suit
  - hand – the cards in the hand (and their order)
  - deck – the cards in the deck (and their order)

https://freesvg.org/nine-of-spades-playing-card-vector-illustration
https://commons.wikimedia.org/wiki/File:Hand_of_cards.jpg
https://www.publicdomainpictures.net/en/view-image.php?image=11873&picture=deck-of-cards

Consider the classic board game Scrabble. (Click on the link to read more about if you aren't familiar with it.) If you were going write a program for Scrabble, what would be classes in your program? Choose all that apply.

challenge	2 respondents	18 %
consonant		0 %
draw tile	2 respondents	18 %
board square	5 respondents	45 % ✓
letter	3 respondents	27 %
game board	11 respondents	100 % ✓
tile rack	7 respondents	64 % ✓
tile	9 respondents	82 % ✓
tile bag	6 respondents	55 % ✓
turn	3 respondents	27 %
scoring	1 respondent	9 %
scrabble dictionay (legal words)	9 respondents	82 % ✓
score	4 respondents	36 %
player	11 respondents	100 % ✓
point value		0 %
play word	2 respondents	18 %
vowel	1 respondent	9 %

the correct answers are all nouns – in most cases, corresponding to physical elements of the game (things you can point to)

https://localscrabble.files.wordpress.com/2011/11/scrabble-1.jpg

Consider the classic board game Scrabble. (Click on the link to read more about if you aren't familiar with it.) If you were going to write a program for Scrabble, what would be classes in your program? Choose all that apply.

challenge	2 respondents	18 %
consonant		0 %
draw tile	2 respondents	18 %
board square	5 respondents	45 % ✓
letter	3 respondents	27 %
game board	11 respondents	100 % ✓
tile rack	7 respondents	64 % ✓
tile	9 respondents	82 % ✓
tile bag	6 respondents	55 % ✓
turn	3 respondents	27 %
scoring	1 respondent	9 %
scrabble dictionay (legal words)	9 respondents	82 % ✓
score	4 respondents	36 %
player	11 respondents	100 % ✓
point value		0 %
play word	2 respondents	18 %
vowel	1 respondent	9 %

not every noun needs to become a class that we write – these types are well-represented by existing Java types

just a `char` value

just an `int` value

just an `int` value

Consider the classic board game Scrabble. (Click on the link to read more about if you aren't familiar with it.) If you were going to write a program for Scrabble, what would be classes in your program? Choose all that apply.

challenge	2 respondents	18 %
consonant		0 %
draw tile	2 respondents	18 %
board square	5 respondents	45 % ✓
letter	3 respondents	27 %
game board	11 respondents	100 % ✓
tile rack	7 respondents	64 % ✓
tile	9 respondents	82 % ✓
tile bag	6 respondents	55 % ✓
turn	3 respondents	27 %
scoring	1 respondent	9 %
scrabble dictionay (legal words)	9 respondents	82 % ✓
score	4 respondents	36 %
player	11 respondents	100 % ✓
point value		0 %
play word	2 respondents	18 %
vowel	1 respondent	9 %

verbs correspond to methods, not classes

verb

verb

Consider the classic board game Scrabble. (Click on the link to read more about if you aren't familiar with it.) If you were going to write a program for Scrabble, what would be classes in your program? Choose all that apply.

challenge	2 respondents	18 %
consonant		0 %
draw tile	2 respondents	18 %
board square	5 respondents	45 % ✓
letter	3 respondents	27 %
game board	11 respondents	100 % ✓
tile rack	7 respondents	64 % ✓
tile	9 respondents	82 % ✓
tile bag	6 respondents	55 % ✓
turn	3 respondents	27 %
scoring	1 respondent	9 %
scrabble dictionay (legal words)	9 respondents	82 % ✓
score	4 respondents	36 %
player	11 respondents	100 % ✓
point value		0 %
play word	2 respondents	18 %
vowel	1 respondent	9 %

sometimes nouns are really verbs

"challenge" can be a noun or a verb, but in context it refers to the player challenging a move – an action (verb)

a noun, but a definition of a kind of letter – determining if a letter is a consonant is a process (verb)

"turn" is a noun, but in context it refers to the player taking actions – it is really the verb "take turn"

sometimes used to refer to a process – "scoring zero"

as a section header, it refers to rules for how to award points

in both cases, "scoring" is really about the action of updating points

a noun, but a definition of a kind of letter – determining if a letter is a vowel is a process (verb)

Gameplay [edit]

A game of *Scrabble* in French

On every turn, the player at turn can perform one of the following options:

- Pass, forfeiting the turn and scoring zero.
- Exchange one or more tiles for an equal number from the bag, scoring zero. This can only be done if 7 or more tiles remain in the bag.
- Play at least one tile on the board, adding the value of all words formed to the player's cumulative score.

The first play of the game must consist of at least two tiles and cover the center square (H8). Any play thereafter must use at least one of the player's tiles to form a "main word" (containing all of the player's played tiles in a straight line) reading left-to-right or top-to-bottom. Diagonal plays are not allowed. At least one tile must be adjacent (horizontally or vertically) to a tile already on the board. If the play includes a blank tile, the player must designate the letter the blank represents; that letter remains unchanged for the rest of the game unless the play is challenged off. The player announces the score for that play, and then draws tiles from the bag equal to the number of tiles played, so that there are seven tiles on their rack. If there are not enough tiles, the player draws any remaining tiles instead. If the game is played using a clock, the player starts the opponent's clock after announcing the score and before drawing tiles. Players may keep track of tiles played during the game.

If a player has made a play and not yet drawn a tile, any other player may choose to challenge any or all words formed by the play. The challenged word(s) are then searched in the agreed-upon word list or dictionary. If at least one challenged word is unacceptable, the play is removed from the board, and the player scores zero for that turn. If all challenged words are acceptable, the challenger loses their turn. In tournament play, players are not entitled to know which word(s) are invalid or the definitions of any challenged words. Penalties for unsuccessfully challenging an acceptable play vary in club and tournament play and are described in greater detail below.

https://en.wikipedia.org/wiki/Scrabble

Gameplay [edit]

On every turn, the player at turn can perform one of the following options:

- Pass, forfeiting the turn and scoring zero.
- Exchange one or more tiles for an equal number from the bag, scoring zero. This can only be done if 7 or more tiles remain in the bag.
- Play at least one tile on the board, adding the value of all words formed to the player's cumulative score.

A game of *Scrabble* in French

The first play of the game must consist of at least two tiles and cover the center square (H8). Any play thereafter must use at least one of the player's tiles to form a "main word" (containing all of the player's played tiles in a straight line) reading left-to-right or top-to-bottom. Diagonal plays are not allowed. At least one tile must be adjacent (horizontally or vertically) to a tile already on the board. If the play includes a blank tile, the player must designate the letter the blank represents; that letter remains unchanged for the rest of the game unless the play is challenged off. The player announces the score for that play, and then draws tiles from the bag equal to the number of tiles played, so that there are seven tiles on their rack. If there are not enough tiles, the player draws any remaining tiles instead. If the game is played using a clock, the player starts the opponent's clock after announcing the score and before drawing tiles. Players may keep track of tiles played during the game.

If a player has made a play and not yet drawn a tile, any other player may choose to challenge any or all words formed by the play. The challenged word(s) are then searched in the agreed-upon word list or dictionary. If at least one challenged word is unacceptable, the play is removed from the board, and the player scores zero for that turn. If all challenged words are acceptable, the challenger loses their turn. In tournament play, players are not entitled to know which word(s) are invalid or the definitions of any challenged words. Penalties for unsuccessfully challenging an acceptable play vary in club and tournament play and are described in greater detail below.

Gameplay [edit]

On every turn, the player at turn can perform one of the following options:

- Pass, forfeiting the turn and scoring zero.
- Exchange one or more tiles for an equal number from the bag, scoring zero. This can only be done if 7 or more tiles remain in the bag.

French

Scoring [edit]

The score for a play is determined as follows:

- The value of each tile is indicated with a point value (between 1 and 10, with blanks worth zero points), and the score of every new word formed is equal to the sum of the point values of the letters in that word. If a play covers any premium squares (such as DLS or TWS squares), the point value of the corresponding letter or word is multiplied by 2 or 3 respectively. The center star is also a DWS square.
- Premium squares only apply when newly placed tiles cover them. Any subsequent plays do not count these premium squares. A play that covers a DWS or TWS multiplies the value of the entire word(s) by 2 or 3, including tiles already on the board.
- If a newly placed word covers both letter and word premium squares, the letter premium(s) is/are calculated first, followed by the word premium(s).
- If a player makes a play where the main word covers two DWS squares, the value of that word is doubled, then redoubled (i.e. 4× the word value). Similarly, if the main word covers two TWS squares, the value of that word is tripled, then re-tripled (9× the word value). Such plays are often referred to as "double-doubles" and "triple-triples" respectively.
- If a player plays all seven of their tiles on their turn (known as a "bingo" in North America and as a "bonus" elsewhere), a 50-point bonus is added to the score of the play.

Premium square colors

Square	Original version	Mattel version (2020-)	Mattel version (2012–2020)	Hasbro Version (2008–2014)
Double letter (DLS)	Light blue	Dark blue	Light blue	Bright blue
Triple letter (TLS)	Blue	Hot pink	Blue	Green
Double word (DWS)	Pink	Yellow	Yellow	Red
Triple word (TWS)	Red	Light green	Red	Orange

Scoreless turns can occur when a player passes, exchanges tiles, loses a challenge, or otherwise makes an illegal move. A scoreless turn can also occur if a play consists of only blank tiles, but this is extremely unlikely in actual play.

Implementing Classes

- *program design* refers to deciding on classes and their methods
- implementing a class requires deciding what to store (instance variables) and how to store it (type)
  - as a collection
    - a concrete data structure – array, linked list, tree
    - from the Java Collections Framework – List, Stack, Queue, PriorityQueue, Map, Set
    - another collection type – e.g. BinaryTree, Trie, PrefixTree
  - as a single variable of some existing type
  - as an object made up of one or more single variables and collections – a new type

Choosing an Implementation

How to choose between different implementations?

- consider the properties of your data, the manipulations you need, and the semantics of the available types

Look for:

- a *logical match* – a type whose concept matches the properties of your thing, and which supports at least the operations you need (and ideally not too many more)
  - write your own class if the only choices have too many operations that aren't relevant – but you could still save effort by using one of those choices to implement your class
- *efficiency* for the operations you'll use
  - more critical for large quantities of data
  - more critical for frequently-used operations
- *ease of implementation* – a slightly less efficient solution can be worthwhile if you can utilize existing code
  - especially for prototyping
  - hide the choice inside a class (private instance variable or helper method) or method (in the method body) for easy changing later

Choosing Between Collections ADTs

Use a queue when –
- you want things out in the same order you put them in

Use a priority queue when –
- you want to remove things in sorted order but you don't necessarily have all of the things at the beginning

Use a stack when –
- you want things out in the reverse of the order you put them in
- you want to access the most recent thing added

Use a list when –
- stacks and queues don't serve your needs
- need to insert/remove/access at any position

Use a dictionary when –
- you want to associate values with keys and do efficient lookup

Use a set when –
- you want to ask questions (only) about membership

Programming With Collections

- use the JCF class matching the ADT whenever possible
  - in variable, parameter, and type declarations
  - `List`, `Stack`, `Queue`, `PriorityQueue`, `Map`, `Set`
- use a specific implementation only when creating new objects
  - choose based on efficiency for the operations that you need to use or that will be used most often
  - List: `ArrayList`, `LinkedList`
    - `ArrayList` for rank-based operations, `LinkedList` only if rank-based operations are not or are only rarely needed
  - Queue: `ArrayDeque`, `LinkedList`
    - `LinkedList` for most applications
  - Map: `HashMap`, `TreeMap`
    - `HashMap` unless you need to iterate through the keys in sorted order
  - Set: `HashSet`, `TreeSet`
    - `HashSet` unless you need to iterate through the elements in sorted order

Example

In a typical card game, each player gets a hand of cards. The deck is shuffled and cards are dealt one at a time from the deck and added to the players' hands. In some games, cards can be removed from a hand, and new cards can be added. The game is won or lost depending on the value (ace, 2, 3, ..., king) and suit (spades, diamonds, clubs, hearts) of the cards that a player receives.

- things
  - card
  - hand of cards
  - deck

  each of these has associated info (instance variables) and operations (methods), so they should become classes
- instance variables – relevant information about the things
  - card – value, suit
  - hand – the cards in the hand (and their order)
  - deck – the cards in the deck (and their order)

  value and suit are simple things (each is a single value per card) but the possible values – value includes "ace" and "king", suit is "spades", "diamonds", "clubs", "hearts" – don't match existing Java types

  the best solution here is to define `Value` and `Suit` as enums – an enum defines a new type by its set of allowed values – but if you aren't familiar with enums, integer constants (e.g. 0 means spades, 1 means diamonds, etc) are an alternative
- methods – access/manipulation of the stored info
  - card – get value, get suit
  - hand – add card, add card at a position, get card at a position, remove card at a position
  - deck – shuffle, deal card

  for the cards in the hand, the order is determined by things external to the hand and random access (accessing any position at any time) is needed – `List` is the ADT supporting external-to-the-collection ordering, and `ArrayList` is the implementation that supports random access

Example

In a typical card game, each player gets a hand of cards. The deck is shuffled and cards are dealt one at a time from the deck and added to the players' hands. In some games, cards can be removed from a hand, and new cards can be added. The game is won or lost depending on the value (ace, 2, 3, ..., king) and suit (spades, diamonds, clubs, hearts) of the cards that a player receives.

- things
  - card
  - hand of cards
  - deck
- instance variables – relevant information about the things
  - card – value, suit
  - hand – the cards in the hand (and their order)
  - deck – the cards in the deck (and their order)

  for the cards in the deck, the order is determined by the deck and random access is needed for shuffling – `List` is the ADT supporting external-to-the-collection ordering, and `ArrayList` is the implementation that supports random access
- methods – access/manipulation of the stored info
  - card – get value, get suit
  - hand – add card, add card at a position, get card at a position, remove card at a position
  - deck – shuffle, deal card

  the getters need no parameters and return a value or suit, which will be of the same type as the corresponding instance variables

```
int getValue ()
int getSuit ()
```

  a card will be of type `Card`, and positions are naturally numbered by integers

```
void addCard ( Card card )
void addCard ( Card card, int pos )
Card getCard ( int pos )
Card removeCard ( int pos )
```

  neither operation needs parameters, and a card will be of type `Card`

```
void shuffle ()
Card dealCard ()
```