OOP Foundations

Make a Klondike solitaire game.

(focus on the game engine parts – everything but the UI, but should be able to support eventually adding UI)

Klondike Solitaire

The Layout

- **Stock** — the face-down draw pile (starts with the 24 cards not dealt to the tableau)
- **Waste** — face-up pile where cards from the stock are turned up to become playable
- **Tableau** — 7 columns, dealt with 1, 2, 3, 4, 5, 6, 7 cards respectively (left to right); only the **top card of each column starts face-up**, the rest face-down
- **Foundations** — 4 piles (one per suit), built up from Ace to King; the win condition is all 52 cards moved here

Dealing

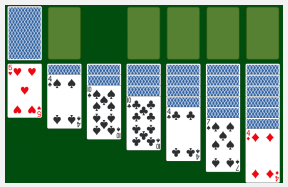
- Shuffle a standard 52-card deck
- Deal 7 columns as above (28 cards total), remaining 24 cards form the stock

Legal Moves

- **Stock → Waste:** flip the top card of the stock face-up onto the waste (when the stock is empty, the waste is turned back over to re-form the stock — some variants allow this only once, others unlimited;)
- **Waste/Tableau → Foundation:** a card may move to its suit's foundation only if it's the next rank up (Ace first, then 2, 3, … King)
- **Waste/Tableau → Tableau:** a card (or a face-up sequence) may move onto another tableau column only if it's **one rank lower and the opposite color** of the card it's placed on (e.g., black 7 onto red 8)
- **Empty tableau column:** only a **King** (or a face-up sequence starting with a King) may be placed on an empty column
- **Flipping:** whenever a move exposes a face-down card at the top of a tableau column, that card is turned face-up

Win Condition

- All 52 cards successfully moved to the 4 foundations, In order, by suit



Object-Oriented Programming (OOP)

- a program has
  - a bunch of variables that store values
  - a bunch of instructions that access/manipulate those variables
- to manage large programs, we need organization

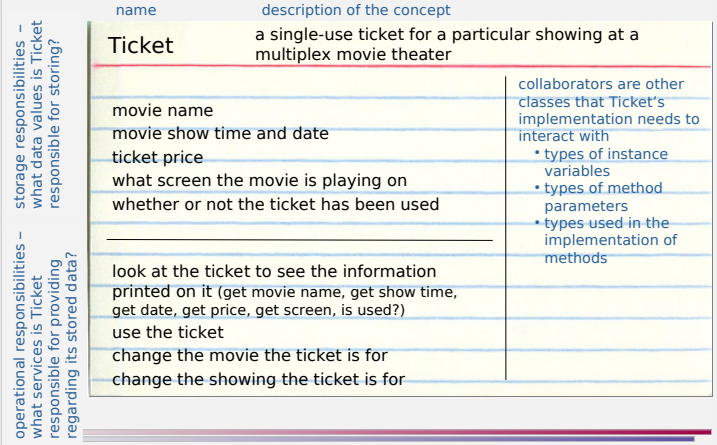
Object-Oriented Programming (OOP)

- in OOP, the main unit of organization is the *object*
  - objects contain data values and operations that work with those values
    - bundles *state* (data values) and *behavior* (methods)
  - objects interact by calling each other's methods
  - *classes* provide definitions for objects
    - the type of each data value (*instance variables*)
    - the signature and body for each operation (*methods*)
    - how to initialize a new instance (*constructor*)
  - a class is a blueprint for a certain kind of object
  - each object is a separate instance of that blueprint

Finding Classes

- objects bundle *related* groups of data values and operations
  - use the concepts already present in the domain as the basis for identifying classes
- strategies and tools
  - textual analysis
    - read through the problem description and identify *candidate nouns* – words that refer to things and concepts
    - not every noun earns a class – can be a judgment call
      - e.g. may refer to different instances rather than different classes
      - e.g. synonyms
      - e.g. may not need to be represented – no associated data values or responsibilities
      - e.g. may not need a new type (e.g. just an `int`)
  - CRC cards
    - captures key info
    - low overhead

CRC Cards – Class / Responsibility / Collaborator



Finding Classes I

this is not a strict linear process – it is useful to focus on classes first since those help to organize everything else but you don't need to ignore storage or operational responsibilities you happen to notice until you get to the proper step

Start with textual analysis –

- textual analysis to identify concepts → classes
  - look for nouns
  - if a concept is a collection, the singular thing may be its own concept
- textual analysis to identify attributes → storage responsibilities
  - a thing is responsible for storing the values of its attributes
  - collections are responsible for storing what's in the collection (the contents)
  - if the order of the elements in the collection matters, the collection is also responsible for storing the order
- textual analysis to identify actions/behaviors → operational responsibilities
  - look for verbs
  - for actions involving info stored by multiple objects, weigh the relevant storage responsibilities and the purpose of the objects' classes
    - assign the operational responsibility to the "most responsible" class
    - if there isn't an appropriate "most responsible" class, assign the responsibility to the class responsible for keeping track of the objects involved
      - if the objects are of the same type, assign to a collections class
    - leave the responsibility unattached if there's not an appropriate class
    - add the other classes involved as collaborators

Finding Classes II

what you write down initially isn't set in stone – don't be afraid to make changes and move stuff around, add things you missed, get rid of stuff that is no longer a good idea, etc as you go along

Then complete the design –

- identify additional storage and operational responsibilities needed to support concepts and actions/behavior already identified
  - e.g. legality checks or responsibilities of collaborators
- address unattached actions/behavior
  - move to an already-identified class if there's a good fit
  - introduce one or more new classes to hold things that still need a home
    - but every class still needs a clear purpose – don't group unrelated unattached things together into a generic Utility class
- throw out stuff you don't need
  - get rid of classes that don't really have any responsibilities or whose only responsibilities are already well-met by something else

first pass – textual analysis to identify concepts → classes

(this includes some responsibilities noticed along the way like standard storage responsibilities – e.g. collections store their elements – and actions/behaviors that are strongly associated with a single class)

class / concept	storage responsibilities	operational responsibilities	collaborators
Stock – stock pile	contents of the pile (the cards and their order)		
Waste – waste pile	contents of the pile (the cards and their order)		
Tableau – the 7 columns	the tableau columns		
TableauColumn – one column within the tableau	contents of the column (the cards and their order)		
Foundation – the in-order pile for one suit	contents of the pile (the cards and their order)		
Card – single playing card	suit, value		
Deck – collection of all 52 cards	contents of the deck (the cards and their order)	shuffle	
CardSequence – one or more in-order face-up cards at the end of a tableau column	contents of the sequence (the cards and their order)		

second pass – textual analysis to identify attributes → storage responsibilities

(new additions highlighted)

class / concept	storage responsibilities	operational responsibilities	collaborators
Stock – stock pile	contents of the pile (the cards and their order)		
Waste – waste pile	contents of the pile (the cards and their order)		
Tableau – the 7 columns	the tableau columns		
TableauColumn – one column within the tableau	contents of the column (the cards and their order)		
Foundation – the in-order pile for one suit	contents of the pile (the cards and their order)		
Card – single playing card	suit, value face up/down		
Deck – collection of all 52 cards	contents of the deck (the cards and their order)	shuffle	
CardSequence – one or more in-order face-up cards at the end of a tableau column	contents of the sequence (the cards and their order)		

third pass – textual analysis to identify actions/behaviors → operational responsibilities (new additions highlighted)

class / concept	storage responsibilities	operational responsibilities	collaborators
Stock – stock pile	contents of the pile (the cards and their order)	restock stock pile from waste pile	Waste
Waste – waste pile	contents of the pile (the cards and their order)		
Tableau – the 7 columns	the tableau columns	move a sequence of cards from one tableau column to another	TableauColumn
TableauColumn – one column within the tableau	contents of the column (the cards and their order)	deal a tableau column	Deck
Foundation – the in-order pile for one suit	contents of the pile (the cards and their order)		
Card – single playing card	suit, value face up/down	flip card face up	
Deck – collection of all 52 cards	contents of the deck (the cards and their order)	shuffle	
CardSequence – one or more in-order face-up cards at the end of a tableau column	contents of the sequence (the cards and their order)		
		single card moves – stock→waste, waste→foundation, etc	multiple classes

while restocking the stock pile and dealing a tableau column involve multiple objects, the operations are more about the stock pile and tableau column – initialization of stored info goes hand-in-hand with the storage responsibility
when moving cards from one pile to another, it doesn't matter where the card being added is coming from or where the card being removed is going to – both piles are on equal footing and there's no reason for one pile to know about any others

identify additional storage and operational responsibilities needed to support concepts and actions/behavior already identified

address unattached actions/behaviors (new additions highlighted)

(see next slide)

added operational responsibilities to collaborators e.g. restocking the stock pile requires removing cards from the waste pile

realized the dealing with moves isn't just actually moving the card(s) from one pile to another – there needs to also be support for checking if a move is legal (for single card moves, this is mostly just a matter of checking if it is legal to add to piles since removing the top card is always legal...as long as the pile isn't empty; for card sequences, it is also necessary to check if it is legal to remove)

realized that the win condition hadn't been accounted for – the game will need to check that

GameEngine added to coordinate the overall game – it keeps track of the various game objects (deck and piles) and is then a natural place for the various moves

class / concept	storage responsibilities	operational responsibilities	collaborators
Stock – stock pile	contents of the pile (the cards and their order)	restock stock pile from waste pile pick up top card is empty	Waste
Waste – waste pile	contents of the pile (the cards and their order)	remove the top card place card on top of the pile is empty	
Tableau – the 7 columns	the tableau columns	move a sequence of cards from one tableau column to another	TableauColumn
TableauColumn – one column within the tableau	contents of the column (the cards and their order)	deal a tableau column pick up the top card pick up the top *n* cards place a card at the top place a card sequence at the top flip the top card face up is it legal to add a card? is it legal to add a card sequence? is empty	Deck
Foundation – the in-order pile for one suit	contents of the pile (the cards and their order)	place card on top of the pile is it legal to add a card? is full	
Card – single playing card	suit, value face up/down	flip card face up get suit, value	
Deck – collection of all 52 cards	contents of the deck (the cards and their order)	shuffle deal a card is empty	
CardSequence – one or more in-order face-up cards at the end of a tableau column	contents of the sequence (the cards and their order)		
GameEngine – coordinate game	the deck and the piles	single card moves – stock→waste, waste→foundation, etc	multiple classes

throw out stuff you don't need (new additions highlighted)

(see next slide)

CardSequence doesn't have any operational responsibilities – do we need it? maybe not – the sequencing rules (decreasing value, alternating color) distinguish a card sequence from an ordinary linear collection like List, but CardSequence is only used as a temporary holder in the implementation of a move from one tableau column to another

Tableau's only real job is storing the tableau columns – with GameEngine now the keeper of the game objects, there's not really a need for the extra layer of grouping that Tableau provides and GameEngine could hold the tableau columns directly

at this point, we should have a sufficient design to proceed – either with further design analysis (such as whether an inheritance hierarchy for the piles make sense) or to start coding

is the design complete? probably not – there may be some internal things (such as instance variables) or smaller things (such as getters) missing, but it should be complete enough to move on – enough of the public structure things are fleshed out so that there shouldn't be any big surprises forcing a redesign

From Design to Code

- card → class
- storage responsibilities → instance variables
- operational responsibilities → methods

- a key thing as you start to implement is to write comments when you write the method headers
  - the design as outlined handles the big picture of what goes where, but it doesn't address details of how individual methods fit together
    - e.g. when a move exposes the top face-down card in a tableau column, that card should be flipped face up – is this an automatic part of picking up cards from the column or it is a separate step done by the move handler?
    - either way, the behavior of the pick up cards method needs to be documented so the caller knows what they need to take care of