Principles of Software Construction: Objects, Design, and Concurrency

Design Patterns

Claire Le Goues Vincent Hellendoorn

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

HW1 grades are released!

- If you have regrade requests, issue them via gradescope within 1 week.

Exam on Thursday; you may bring a cheat sheet 4 pages front and back. No computers/phones/etc.

- Bring writing implements.

Reminder that HW3 is due next Monday.

- Note about diagrams/Justification task.

Note about late days.

A few Santorini things.

The full Santorini rules are available on their website, you can follow the link from the handout.

An “adjacent” square is anywhere in the 8 tiles surrounding another square.

Workers can only move UP one level, but may move DOWN any number of level.

The same worker must move and build.

A player must both move and build in a turn.

Design Patterns

One design scenario

- Amazon.com processes millions of orders each year, selling in 75 countries, all 50 states, and thousands of cities worldwide. These countries, states, and cities have hundreds of distinct sales tax policies and, for any order and destination, Amazon.com must be able to compute the correct sales tax for the order and destination.

Another design scenario

- A vision processing system must detect lines in an image. For different applications the line detection requirements vary. E.g., for a vision system in a driverless car the system must process 30 images per second, but it's OK to miss some lines in some images. A face recognition system can spend 3-5 seconds analyzing an image, but requires accurate detection of subtle lines on a face.

A third design scenario

- Suppose we need to sort a list in different orders…

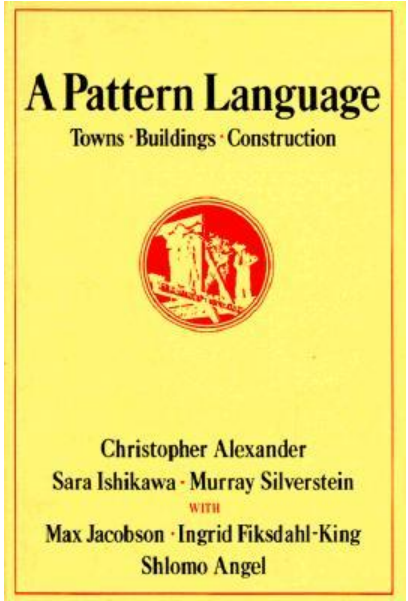
```
const ASC = function(i: number, j: number): boolean {
    return i < j;
}
const DESC = function(i: number, j: number): boolean {
    return i > j;
}

function sort(
    list: number[],
    order: (number, number) => boolean) {
  // …
  boolean mustSwap = order(list[j], list[i]);
  // …
}
```

Design Patterns

Design patterns

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"

– Christopher Alexander, Architect (1977)



How not to discuss design (from Shalloway and Trott)

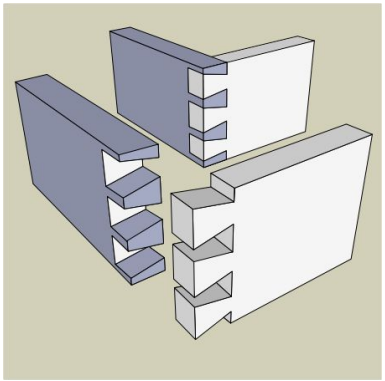
- Carpentry:
  - How do you think we should build these drawers?
  - Well, I think we should make the joint by cutting straight down into the wood, and then cut back up 45 degrees, and then going straight back down, and then back up the other way 45 degrees, and then going straight down, and repeating…

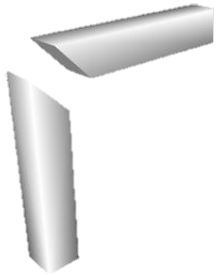
How not to discuss design (from Shalloway and Trott)

- Software Engineering:
  - How do you think we should write this method?
  - I think we should write this if statement to handle … followed by a while loop … with a break statement so that…

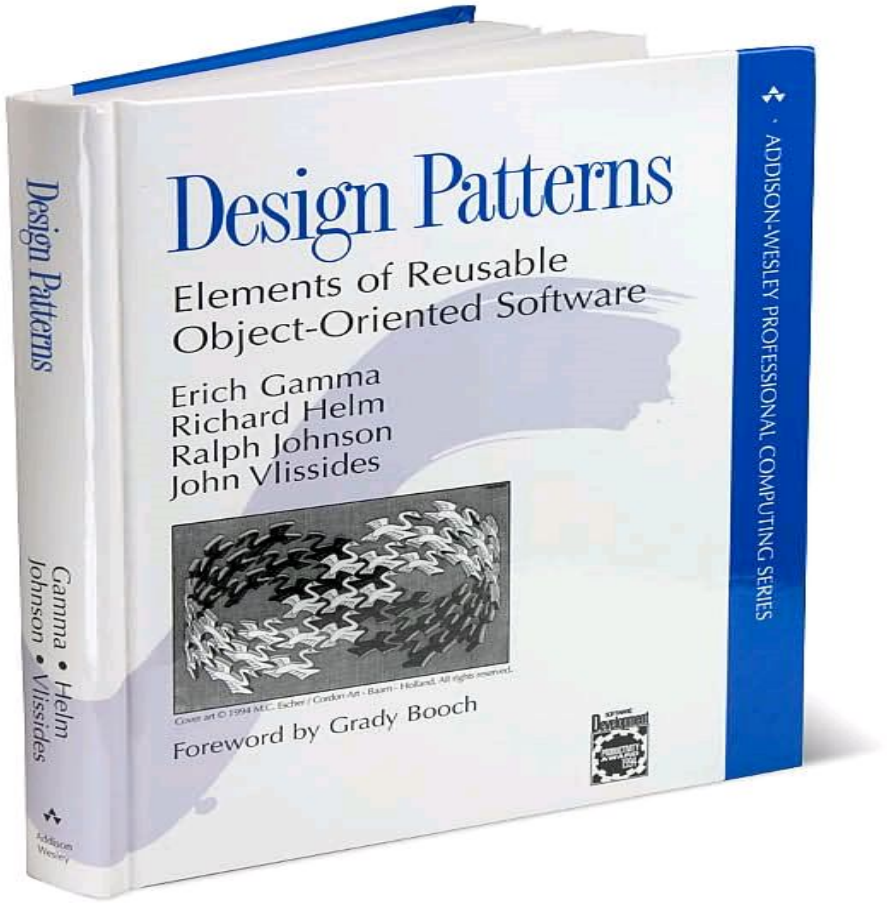
Discussion with design patterns

- Carpentry:
  - "Is a dovetail joint or a miter joint better here?"
- Software Engineering:
  - "Is a strategy pattern or a template method better here?"





History: *Design Patterns* (1994)



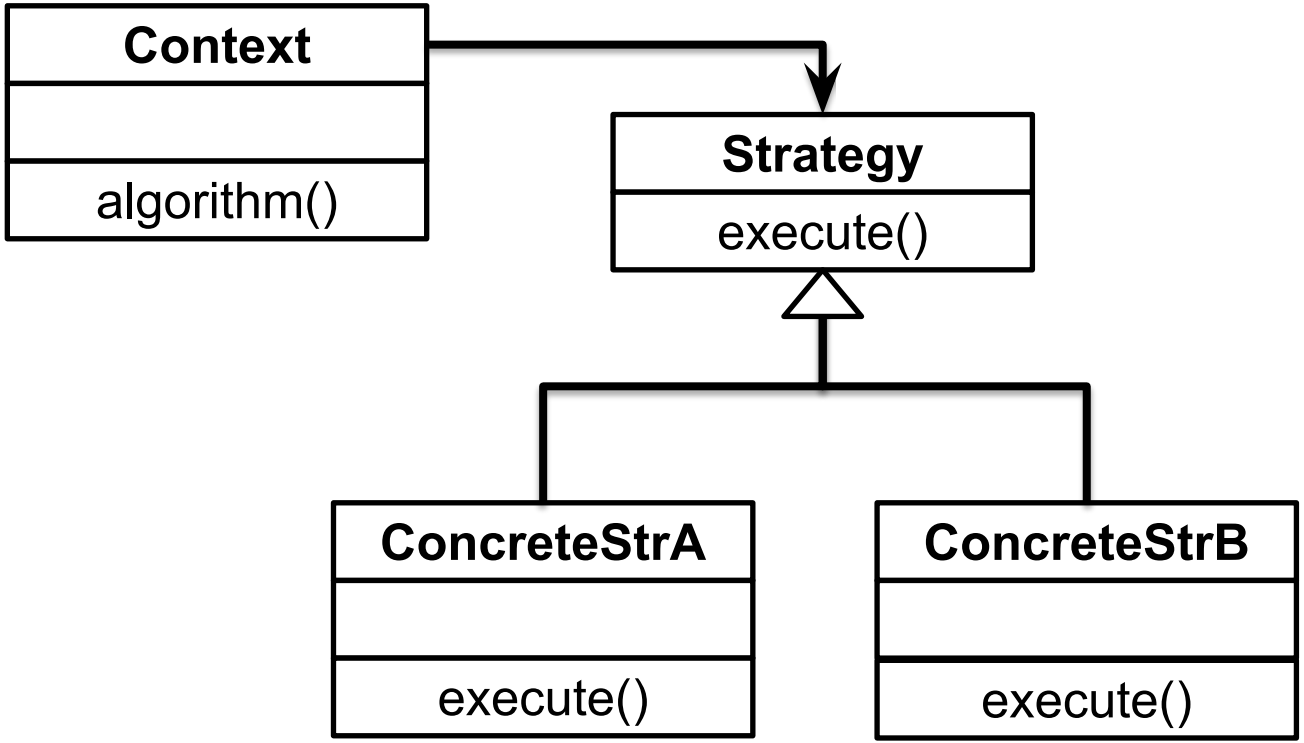
Elements of a design pattern

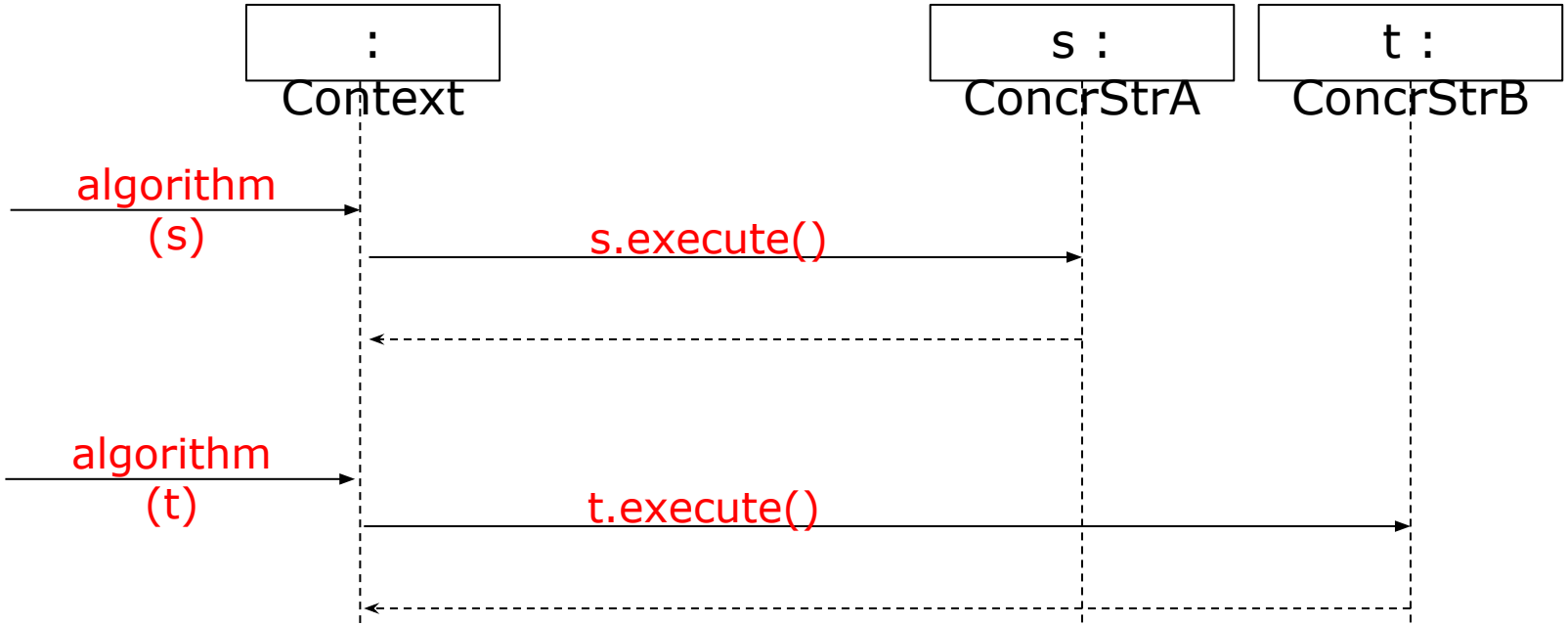
- Name
- Abstract description of problem
- Abstract description of solution
- Analysis of consequences

Strategy Pattern

Strategy pattern

- Problem: Clients need different variants of an algorithm
- Solution: Create an interface for the algorithm, with an implementing class for each variant of the algorithm
- Consequences:
  - Easily extensible for new algorithm implementations
  - Separates algorithm from client context
  - Introduces an extra interface and many classes: (1) Code can be harder to understand, (2) Lots of overhead if the strategies are simple


Context
algorithm()
Strategy
execute()
ConcreteStrA
execute()
ConcreteStrB
execute()



Strategy can be provided in method call or in any other way to context

One design scenario

- Amazon.com processes millions of orders each year, selling in 75 countries, all 50 states, and thousands of cities worldwide. These countries, states, and cities have hundreds of distinct sales tax policies and, for any order and destination, Amazon.com must be able to compute the correct sales tax for the order and destination.

Another design scenario

- A vision processing system must detect lines in an image. For different applications the line detection requirements vary. E.g., for a vision system in a driverless car the system must process 30 images per second, but it's OK to miss some lines in some images. A face recognition system can spend 3-5 seconds analyzing an image, but requires accurate detection of subtle lines on a face.

Design Patterns and Programming Languages

Design patterns address general design challenges

Some patterns address problems with built-in solutions

Example: Strategy pattern vs higher-order functions

```
const ASC = function(i: number, j: number): boolean {
    return i < j;
}
const DESC = function(i: number, j: number): boolean {
    return i > j;
}
```

Strategy Pattern vs Higher-Order Function

```
interface Order {
  boolean lessThan(int i, int j);
}

class AscendingOrder implements Order {
  public boolean lessThan(int i, int j) {
     return i < j; }
}
class DescendingOrder implements Order {
  public boolean lessThan(int i, int j) {
     return i > j; }
}

void sort(int[] list, Order order) ;
```

```
const ASC =
     function(i: number, j: number): boolean {
          return i < j;
     }
const DESC =
     function(i: number, j: number): boolean {
          return i > j;
     }

function sort(
     list: number[],
     order: (number, number) => boolean) ...;
```

Strategy Pattern vs Higher-Order Function

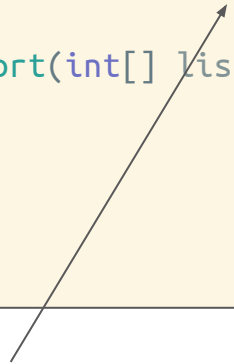
```
interface Order {
  boolean lessThan(int i, int j);
}

class AscendingOrder implements Order {
  public boolean lessThan(int i, int j) {
     return i < j; }
}
class DescendingOrder implements Order {
  public boolean lessThan(int i, int j) {
     return i > j; }
}

void sort(int[] list, Order order) ;
```

```
const ASC = function(i, j) { return i < j; }
const DESC = function(i, j) { return i > j; }

function sort(list, order) ...;
```

New Java Syntax for “Functions”

```
interface Order {
  boolean lessThan(int i, int j);
}

final Order ASCENDING  = (i, j) -> i < j;
final Order DESCENDING = (i, j) -> i > j;

static void sort(int[] list, Order order);
```

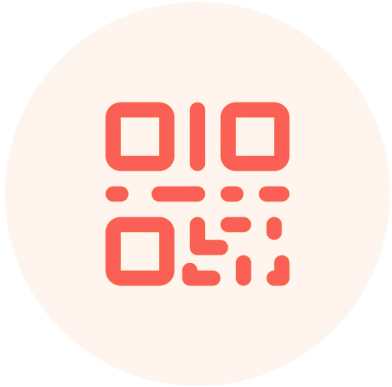
```
const ASC =
     function(i: number, j: number): boolean {
          return i < j;
     }
const DESC =
     function(i: number, j: number): boolean {
          return i > j;
     }

function sort(
     list: number[],
     order: (number, number) => boolean) ...;
```



Convenient syntax (introduced for lambdas) to create objects of interface with single method.

Design alternative practice

slido



Join at slido.com
#1920213

ⓘ Start presenting to display the joining instructions on this slide.

Recall: Blackjack

The objective of Blackjack is to obtain a score higher than the dealer. In each round the player and the dealer are initially dealt 2 cards, with one of the dealer's cards dealt face down. Each numerical card (2-9) is worth its face value, aces are worth either 1 or 11 (it is the player's choice), and all face cards (ten, jack, queen, and king) are worth 10.

At the beginning of each round the player must place a bet; this amount will not change for the rest of the round. After the bet, the player repeatedly decides whether to hit (be dealt a new card, up to 5 cards in the player's hand) or stand (stop being dealt cards). If the player scores more than 21 points, the player loses their bet and the round.

After the player stands, the dealer exposes their face-down card and must hit (be dealt cards) until the value of their cards is more than 16. If the player's score is more than the dealer's final score (but less than or equal to 21) then the player wins an amount equal to their bet. The player also wins if they are dealt 5 cards worth 21 or fewer points.

Otherwise, the player loses their bet. More information can be found on Wikipedia's Blackjack page.

Domain modeling

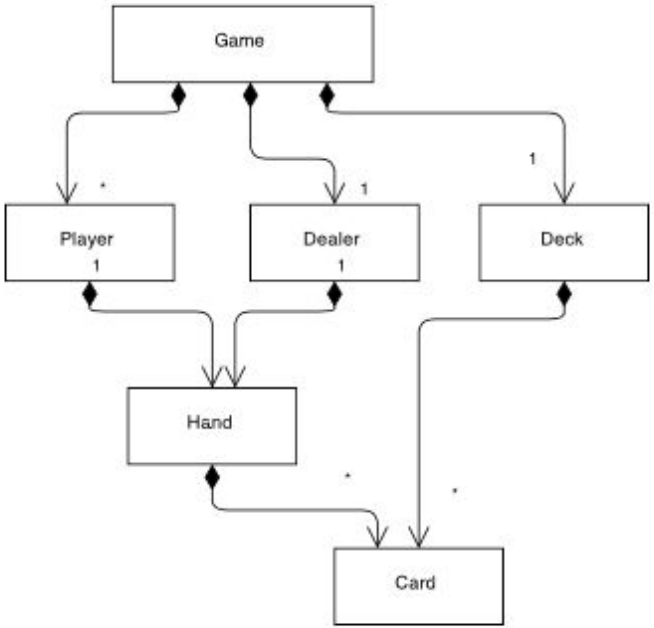
The objective of Blackjack is to obtain a score higher than the dealer. In each round the player and the dealer are initially dealt 2 cards, with one of the dealer's cards dealt face down. Each numerical card (2-9) is worth its face value, aces are worth either 1 or 11 (it is the player's choice), and all face cards (ten, jack, queen, and king) are worth 10.

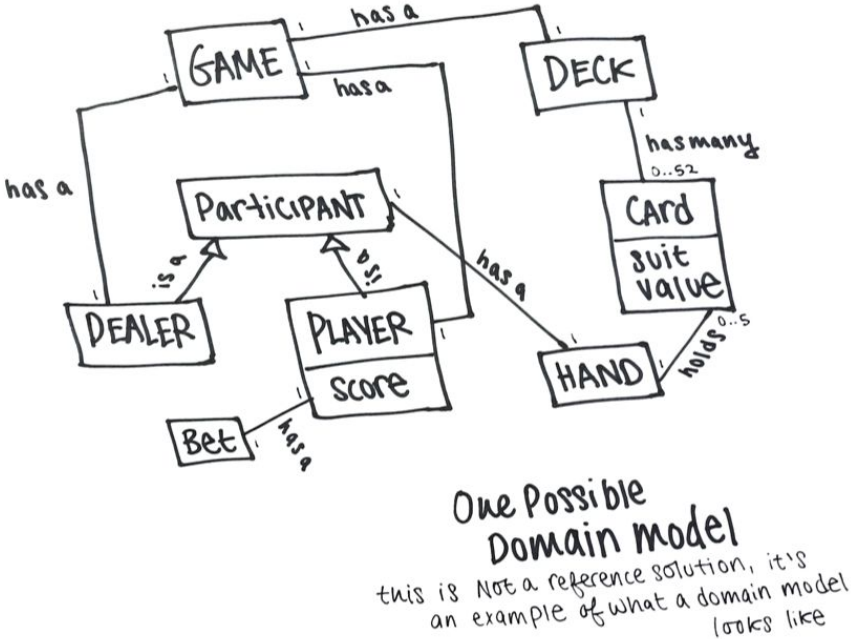
At the beginning of each round the player must place a bet; this amount will not change for the rest of the round. After the bet, the player repeatedly decides whether to hit (be dealt a new card, up to 5 cards in the player's hand) or stand (stop being dealt cards). If the player scores more than 21 points, the player loses their bet and the round.

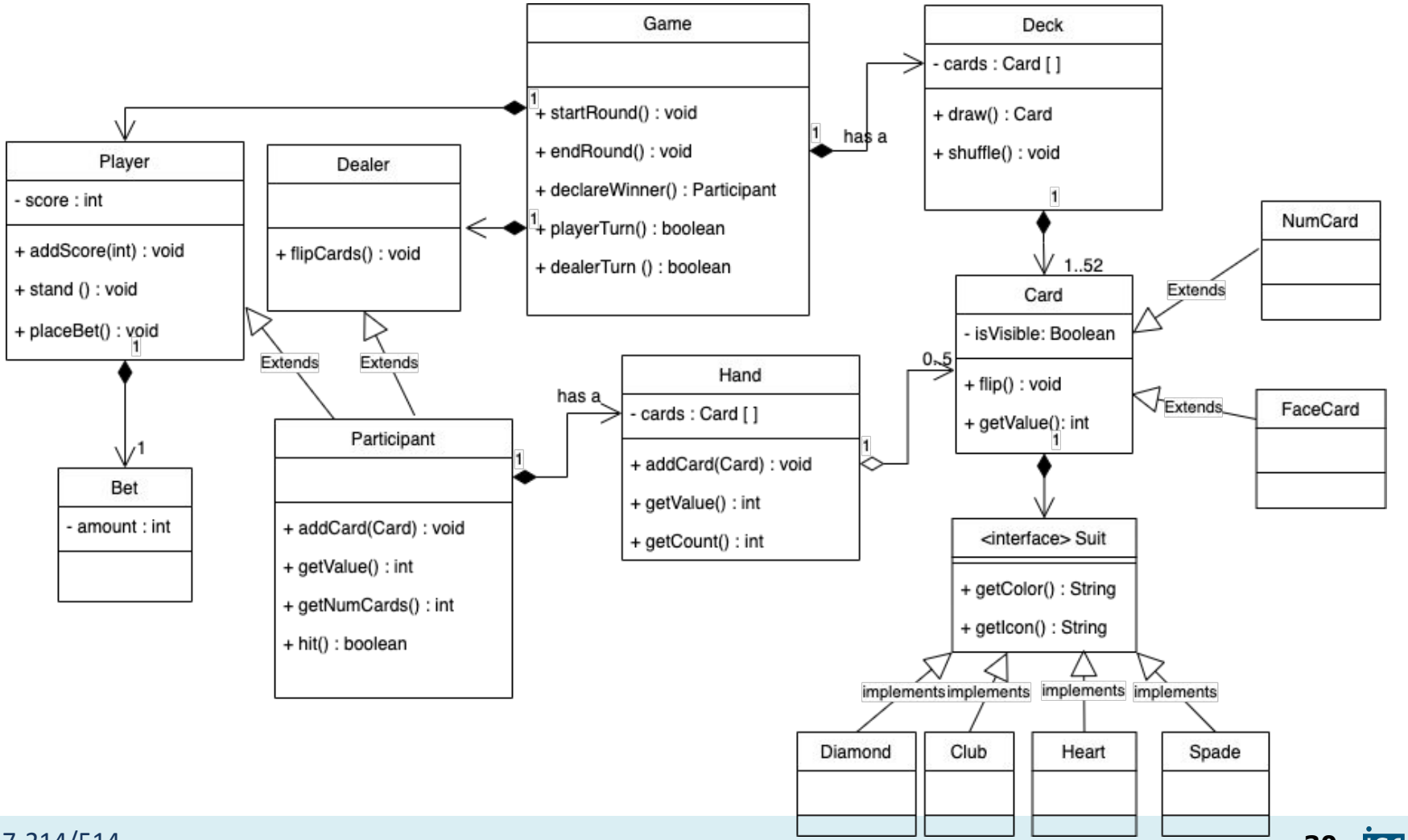
After the player stands, the dealer exposes their face-down card and must hit (be dealt cards) until the value of their cards is more than 16. If the player's score is more than the dealer's final score (but less than or equal to 21) then the player wins an amount equal to their bet. The player also wins if they are dealt 5 cards worth 21 or fewer points.

Otherwise, the player loses their bet. More information can be found on Wikipedia's Blackjack page.

Possible/example domain models






Game
+ startRound() : void
+ endRound() : void
+ declareWinner() : Participant
+ playerTurn() : boolean
+ dealerTurn () : boolean
Deck
- cards : Card [ ]
+ draw() : Card
+ shuffle() : void
has a
Player
- score : int
+ addScore(int) : void
+ stand () : void
+ placeBet() : void
Dealer
+ flipCards() : void
Extends
Extends
Bet
- amount : int
Participant
+ addCard(Card) : void
+ getValue() : int
+ getNumCards() : int
+ hit() : boolean
has a
Hand
- cards : Card [ ]
+ addCard(Card) : void
+ getValue() : int
+ getCount() : int
0..5
1..52
Card
- isVisible: Boolean
+ flip() : void
+ getValue(): int
NumCard
Extends
FaceCard
Extends
<interface> Suit
+ getColor() : String
+ getIcon() : String
implements
implements
implements
implements
Diamond
Club
Heart
Spade

slido

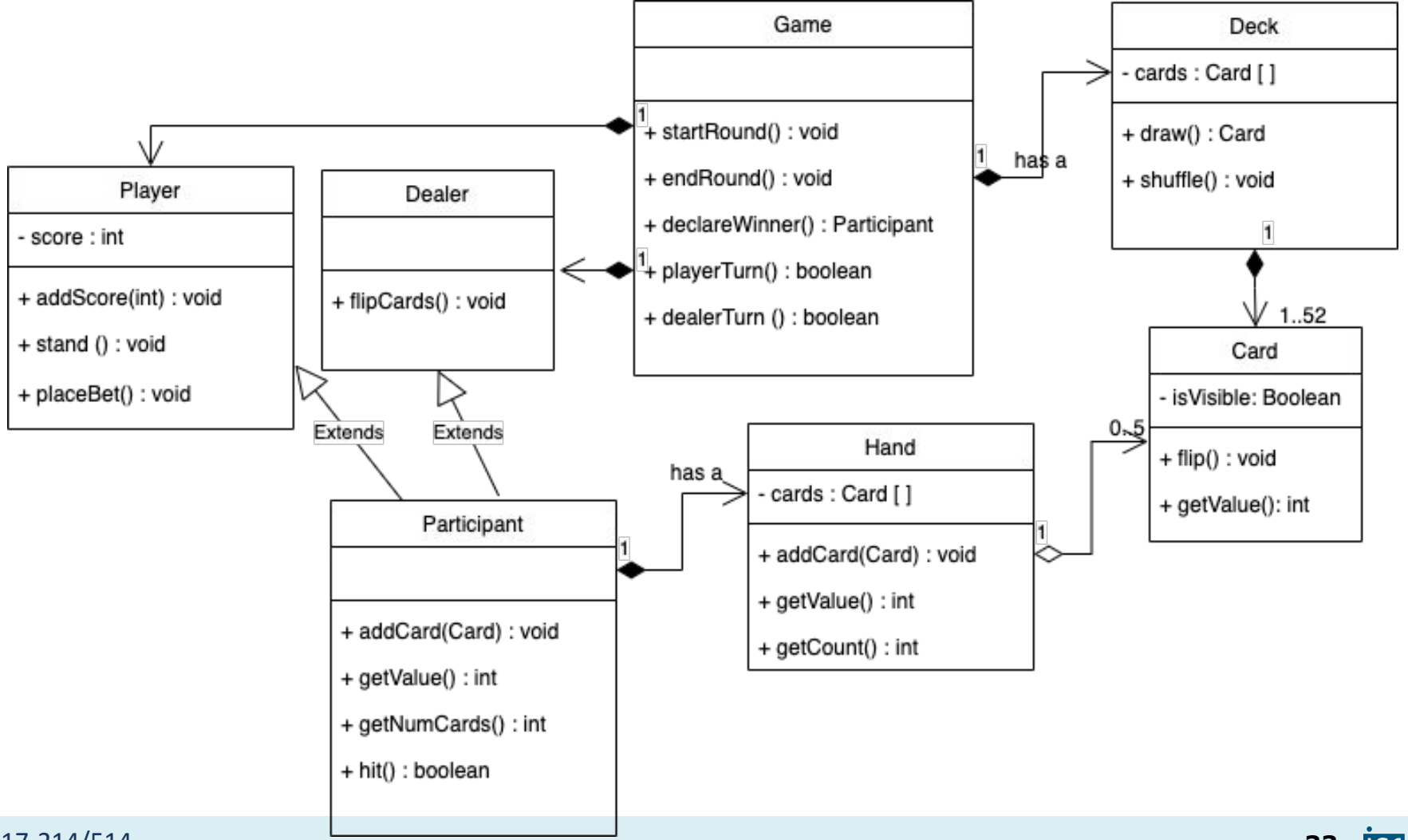
Do you like this design?

ⓘ Start presenting to display the poll results on this slide.

slido

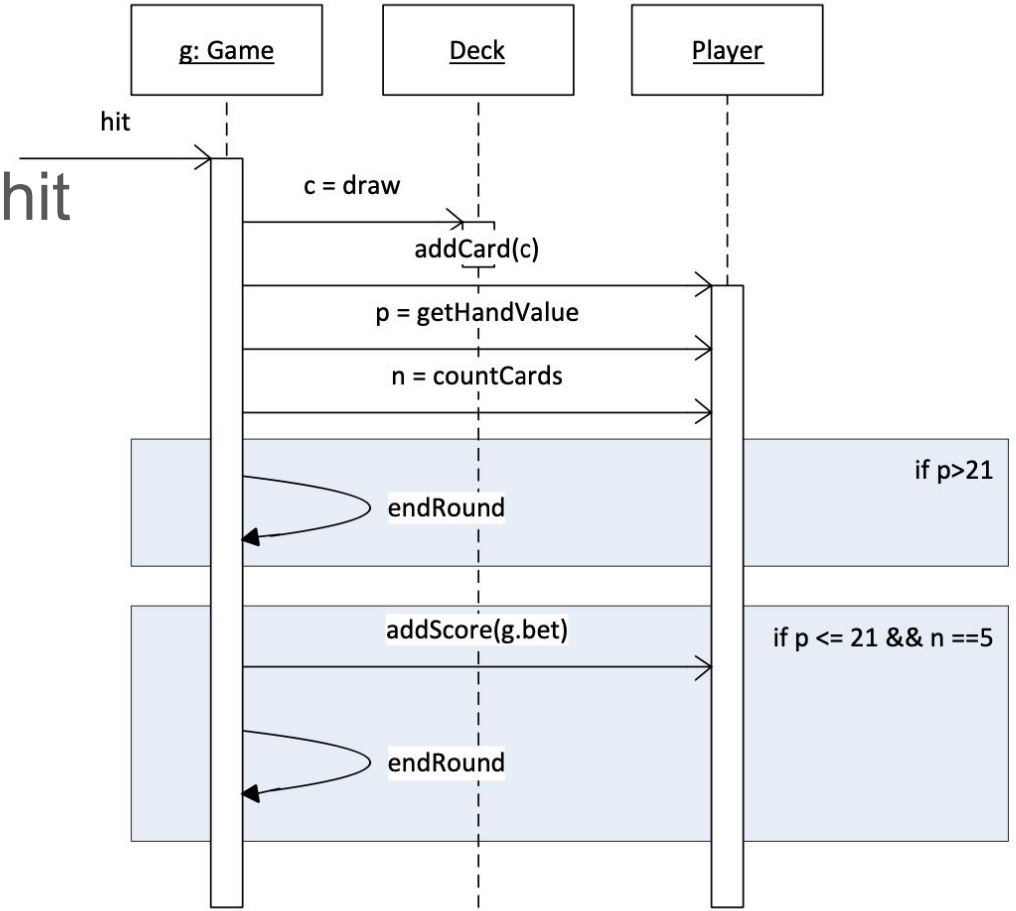
Why/why not?

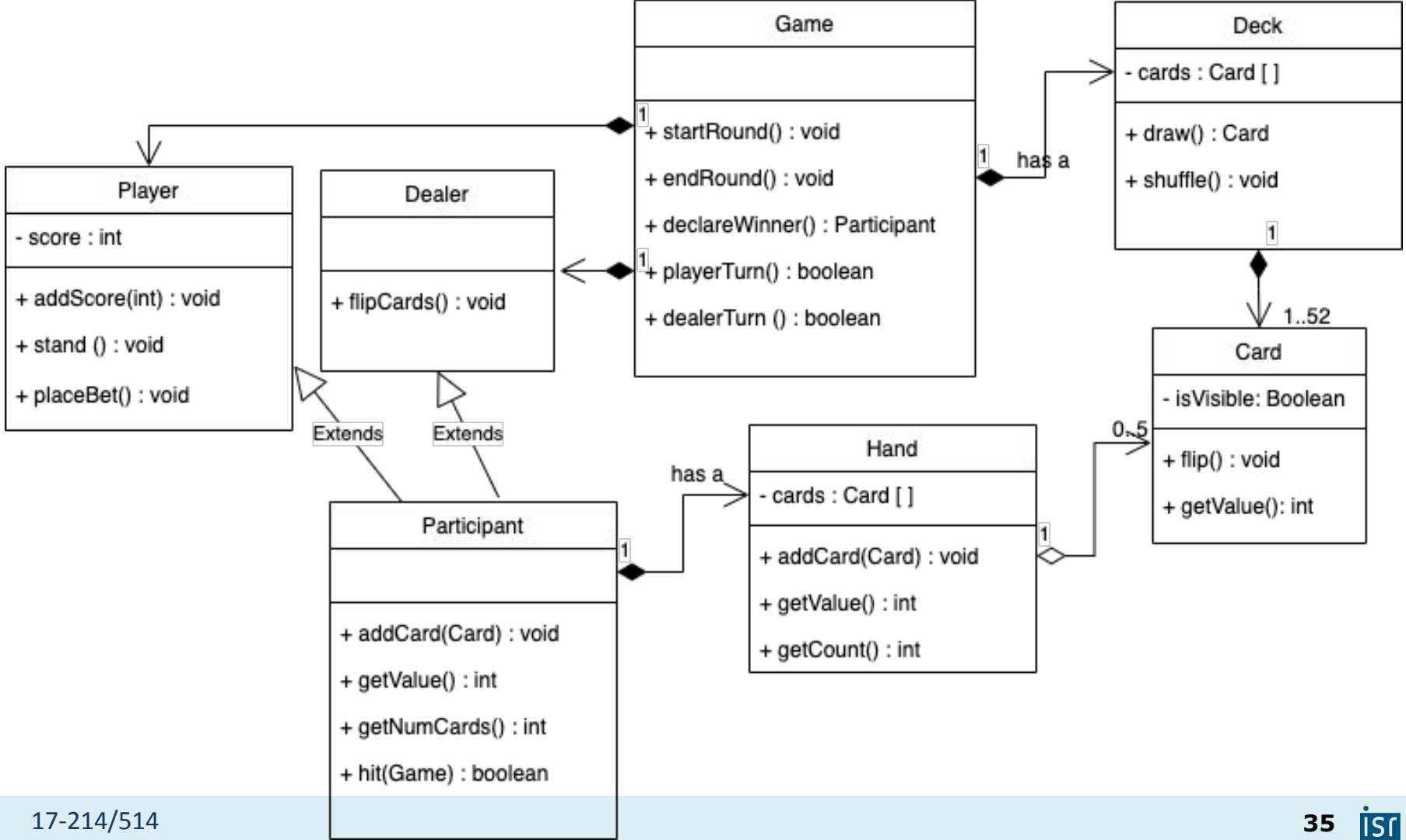
ⓘ Start presenting to display the poll results on this slide.


Game
+ startRound() : void
+ endRound() : void
+ declareWinner() : Participant
+ playerTurn() : boolean
+ dealerTurn () : boolean
Deck
- cards : Card [ ]
+ draw() : Card
+ shuffle() : void
has a
1
1..52
Player
- score : int
+ addScore(int) : void
+ stand () : void
+ placeBet() : void
Dealer
+ flipCards() : void
Extends
Extends
Card
- isVisible: Boolean
+ flip() : void
+ getValue(): int
0..5
Hand
- cards : Card [ ]
+ addCard(Card) : void
+ getValue() : int
+ getCount() : int
has a
Participant
+ addCard(Card) : void
+ getValue() : int
+ getNumCards() : int
+ hit() : boolean

…a reasonable interaction diagram for hit

Now: what does the interaction look like for the object model just now?




Game
+ startRound() : void
+ endRound() : void
+ declareWinner() : Participant
+ playerTurn() : boolean
+ dealerTurn () : boolean
Deck
- cards : Card [ ]
+ draw() : Card
+ shuffle() : void
has a
1
1..52
Player
- score : int
+ addScore(int) : void
+ stand () : void
+ placeBet() : void
Dealer
+ flipCards() : void
Extends
Extends
Card
- isVisible: Boolean
+ flip() : void
+ getValue(): int
0..5
Hand
- cards : Card [ ]
+ addCard(Card) : void
+ getValue() : int
+ getCount() : int
has a
Participant
+ addCard(Card) : void
+ getValue() : int
+ getNumCards() : int
+ hit(Game) : boolean

slido

How's this one going?

ⓘ Start presenting to display the poll results on this slide.

Warning: beware of turning Game into a God Class!

(I struggled to make an example version of Blackjack that violated this principle without being too obvious.)

More design patterns

Module Pattern

Module pattern: Hide internals in closure

```
(function () {
    // ... all vars and functions are in this scope only
    // still maintains access to all globals
}());
```

Function provides local scope, internals not accessible

Function directly invoked to execute it once

Wrapped in parentheses to make it expression

Discovered around 2007, became very popular, part of Node

Using closures to hide methods and fields

```
function createPolarPoint(len, angle) {
    let xcache = -1;
    let internalLen=len;
    function computeX() {…}
    return {
        getX: function() {
            computeX(); return xcache; },
        getY: function() {
            return len * sin(angle); }
    };
}
```

Module pattern: Decide what to export

```
var MODULE = (function () {
    var my = {},
    privateVariable = 1;

    function privateMethod() {
        // ...
    }

    my.moduleProperty = 1;
    my.moduleMethod = function () {
        // ...
    };

    return my;
}());
```

Java: Module Pattern?

Public/private built in, problem does not exist

Fully qualified names (“edu.cmu.cs17214.FlashCard”) as convention/pattern to solve naming clashes

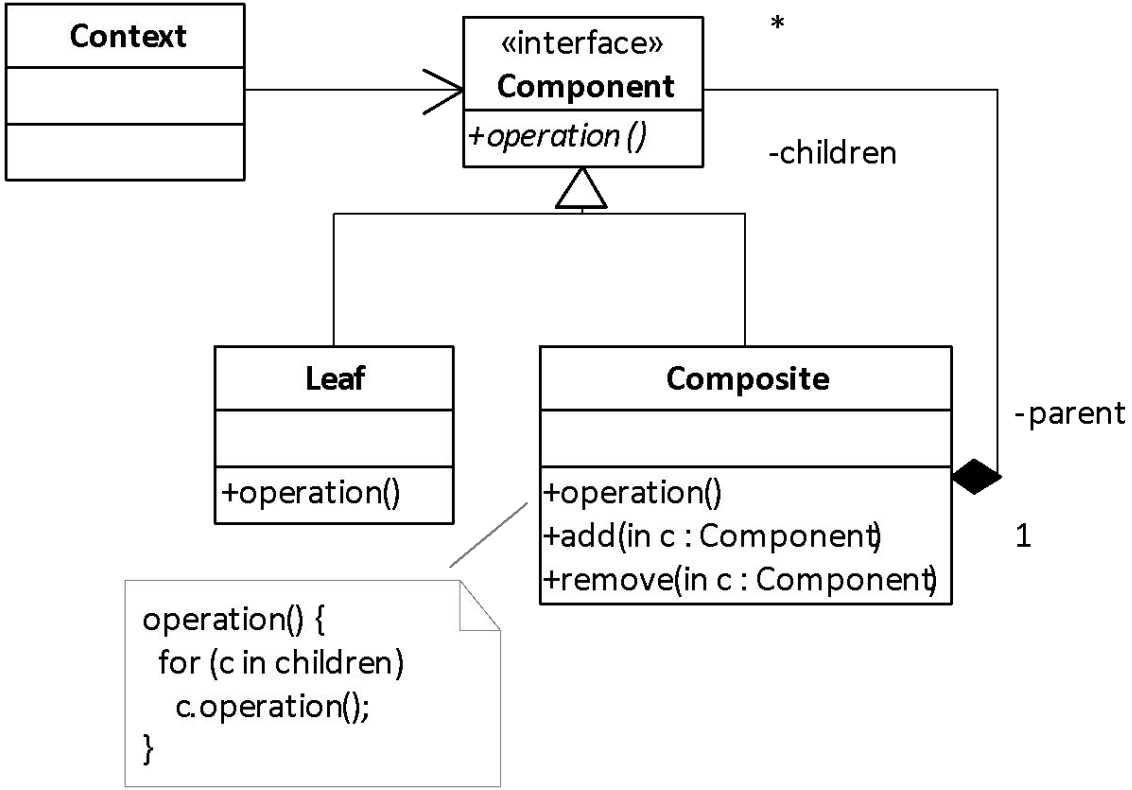
Newer JavaScript/TypeScript features make it less important (ES6 modules, classes, public/private)

Composite Pattern

Design Exercise (on paper)

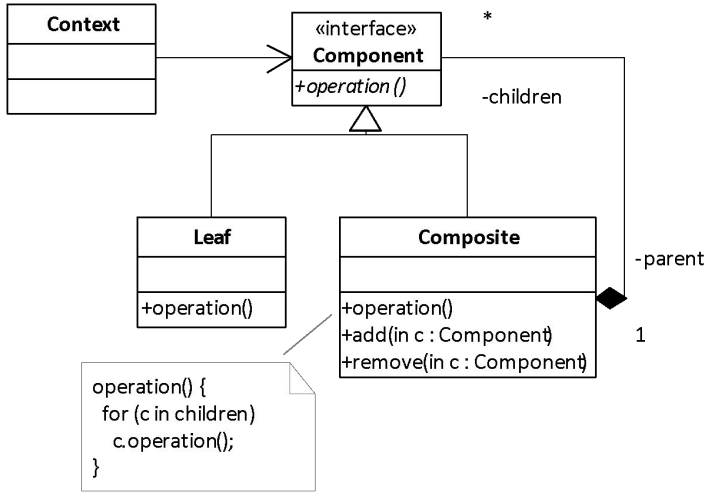
- You are designing software for a shipping company.
- There are several different kinds of items that can be shipped: letters, books, packages, fragile items, etc.
- Two important considerations are the **weight** of an item and its **insurance cost**.
  - Fragile items cost more to insure.
  - All letters are assumed to weigh an ounce
  - We must keep track of the weight of other packages.
- The company sells **boxes** and customers can put several items into them.
  - The software needs to track the contents of a box (e.g. to add up its weight, or compute the total insurance value).
  - However, most of the software should treat a box holding several items just like a single item.
- Think about how to represent packages; what are possible interfaces, classes, and methods? (letter, book, box only)

The Composite Design Pattern



The Composite Design Pattern

- Applicability
  - You want to represent part-whole hierarchies of objects
  - You want to be able to ignore the difference between compositions of objects and individual objects
- Consequences
  - Makes the client simple, since it can treat objects and composites uniformly
  - Makes it easy to add new kinds of components
  - Can make the design overly general
    - Operations may not make sense on every class
    - Composites may contain only certain components



We have seen this before

```
interface Point {
    int getX();
    int getY();
}
class MiddlePoint implements Point {
    Point a, b;
    MiddlePoint(Point a, Point b) {this.a = a; this.b = b; }
    int getX() { return (this.a.getX() + this.b.getX()) / 2;}
    int getY() { return (this.a.getY() + this.b.getY()) / 2; }
}
```

Composite Pattern and Flash Cards?

We have seen this before

```
function newCombinedCardOrganizer (cardOrganizers: CardOrganizer[]): CardOrganizer {
 return {
   reorganize: function (cards: CardStatus[]): CardStatus[] {
     let status = cards.slice()
     for (const cardOrganizer of cardOrganizers) {
       status = cardOrganizer.reorganize(status)
     }
     return status
   }
 }
}
```

Design pattern conclusions

- Provide shared language
- Convey shared experience
- Can be system and language specific