Where we are

• Basics of Reasoning about code
• Coming up
  – Specification: What are we supposed to build?
  – Design: Abstraction. Which designs are “better”?
  – Implementation: Building code to meet a specification
  – Testing: Systematically finding problems
  – Debugging: Systematically fixing problems
  – Maintenance: How does the artifact adapt over time?
  – Documentation: What do we need to know to do these things? How/where do we write that down?

Class Interface

class MutableList<T : Comparable> {
  var count : Int
  func get(index: Int) -> T { ... }
  func set(index: Int, to value: T) -> T { ... }
  func append(_ t : T) { ... }
  ...
  static func isSubsequence(_ part : MutableList<T>, of list: MutableList<T>) -> Bool {
    ...
  }
}
Just Read The Code

```swift
static func isSubsequence(_ part : MutableList<T>, of list : MutableList<T>) -> Bool {
    var partIndex = 0
    for element in list {
        if element == part.get(partIndex) {
            partIndex += 1
            if partIndex == part.count {
                return true
            }
        } else {
            partIndex = 0
        }
    }
    return false
}
```

Just Read The Comments

```swift
// Check whether part appears as a contiguous subsequence of list.
static func isSubsequence(_ part : MutableList<T>, of list : MutableList<T>) -> Bool {
    var partIndex = 0
    for element in list {
        if element == part.get(partIndex) {
            partIndex += 1
            if partIndex == part.count {
                return true
            }
        } else {
            partIndex = 0
        }
    }
    return false
}
```

Write Appropriate Specification

```swift
// Check whether part appears as a contiguous subsequence of list.

• Document Caveats
  // * If list is empty, always returns false
  // * Results may be unexpected if partial matches can happen right before a real match; e.g., (1,2,1,3) will not be identified as a sub sequence of (1,2,1,2,1,3).

• Or Replace with More Detailed Behaviour
  // This method scans “list” from beginning to end, building up a match for “part”, and resetting that match every time that...
```

Write Better Code... (And Spec)

```swift
// Returns true iff there exist possibly empty sequences A, B where
// list = A : part : B
// and “:” is sequence concatenation.
static func isSubsequence(_ part : MutableList<T>, of list : MutableList<T>) -> Bool {
    ...
    }
```
Swift Comments

```swift
/**
 * Returns the first index where the specified value appears in the collection.
 *
 * After using `firstIndex(of:)` to find the position of a particular element in a collection, you can use it to access the element by subscripting. This example shows how you can modify one of the names in an array of students.
 *
 * ```swift
 * var students = ["Ben", "Ivy", "Jordell", "Maxime"]
 * if let i = students.firstIndex(of: "Maxime") {
 *     students[i] = "Max"
 * }
 * print(students)
 * // Prints "["Ben", "Ivy", "Jordell", "Max"]"
 * ```
 *
 * - **Complexity**: O(n), where n is the length of the collection.
 * - **Parameters**
 *     - `element`: An element to search for in the collection.
 * - **Returns**
 *     - The first index where `element` is found. If `element` is not found in the collection, returns nil.
 */

func firstIndex(of element: Element) -> Int?
```

CS326 Specifications

```swift
/**
 * Requires**: none (can omit in this case)
 * Modifies**: self
 * Effects**: Changes the first occurrence of oldValue to newValue
 *
 * ```swift
 * func replace(_ oldValue: T, with newValue: T) -> Int? {
 *     for i in 0..<count {
 *         if get(i) == oldValue {
 *             set(i, to: newValue)
 *             return i
 *         }
 *     }
 *     return nil
 * }
 * ```
 */
```
CS326 Specifications

/**
 * **Requires**: list1 and list2 are the same size
 * **Modifies**: list1
 * **Effects**: the ith element of other is added to the ith element of self
 */
func add(_ list1: MutableList<Int>, _ list2 : MutableList<Int>) {
    for i in 0..<count {
        list1.set(i, list1.get(i) + list2.get(i))
    }
}
Satisfaction of a Specification

- Let \( M \) be an implementation and \( S \) a specification

- \( M \) satisfies \( S \) if and only if
  - Every behavior of \( M \) is permitted by \( S \)

- If \( M \) does not satisfy \( S \), either (or both!) could be “wrong”
  - Usually better to change the program than the spec

Comparing Specifications

- Specification \( S_1 \) is weaker than \( S_2 \), if for all \( M \),
  \[
  \text{\( M \) satisfies \( S_2 \)} \implies \text{\( M \) satisfies \( S_1 \)}
  \]

- A weaker specification gives greater freedom to the implementer

Which is Weaker? A or B?

```plaintext
func index(of element: Element) -> Int? {
    for i in 0..<count {
        if get(i) == element {
            return i
        }
    }
    return nil
}
```

Specification A
- requires: value occurs in self
- returns: \( i \) such that \( \text{get}(i) = \text{value} \)

Specification B
- requires: value occurs in self
- returns: smallest \( i \) such that \( \text{get}(i) = \text{value} \)

Which is Weaker? A or C?

```plaintext
func index(of element: Element) -> Int? {
    for i in 0..<count {
        if get(i) == element {
            return i
        }
    }
    return nil
}
```

Specification A
- requires: value occurs in self
- returns: \( i \) such that \( \text{get}(i) = \text{value} \)

Specification C
- returns: \( i \) such that \( \text{get}(i) = \text{value} \), or \( \text{nil} \) if value is not in self

Weaker Specification:
- Implementer: Easier to satisfy (more implementations satisfy it)
- Client: Harder to use (fewer guarantees)
Weakening a Specification

- Promise Less
  - Weaker Postcondition
    - Returns clause easier to satisfy
    - More objects in modifies clause
    - Effects clause easier to satisfy
    - Fewer specific exceptions
- Ask more of client
  - Stronger Precondition
    - Requires clause harder to satisfy

(Strengthening: The Opposite)

Stronger and Weaker Specifications

- Weaker specification:
  - Implementer: Easier to satisfy (more implementations satisfy it)
  - Client: Harder to use (fewer guarantees)
- Stronger specification:
  - Implementer: Harder to satisfy
  - Client: Easier to use (more guarantees, more predictable, can make more assumptions)

Which is Better?

- Stronger does not always mean better!
- Weaker does not always mean better!
- Strength of specification trades off:
  - Usefulness to client
  - Ease of simple, efficient, correct implementation
  - Promotion of reuse and modularity
  - Clarity of specification itself
- “It depends”

Two Representations of Points

class Point {
    public float x;
    public float y;
    public float theta;
}

```java
class Point {
    public float x;
    public float y;
    public float theta;
}
```
Point ADT

```java
public class Point {
    // A 2-d point exists in the plane, ...
    public var x : Double
    public var y : Double
    public var r : Double
    public var theta : Double
    // ... can be created, ...
    public init() // new point at (0,0)
    public init(points : Set<Point>) // centroid
    // ... can be moved, ...
    public func translate(dx: Double, dy: Double)
    public func scaleAndRotate(dr: Double, dTheta: Double)
}
```

Abstract Data Type = Objects + Ops

```
Clients

rest of program

Point
x
y
r
theta
translate
scaleRot

Observers – may be actual or computed properties.

Creators/Producers

 Implementors

abstraction barrier

Mutators

Poly: Overview and Abstract State

```java
/**
 * A Poly is an immutable polynomial with integer coefficients. A typical Poly is
 * \[c_0 + c_1 * x + c_2 * x^2 + \ldots\]
 */
public class Poly {
    // Abstract state (specification fields)
}```

Poly: Creators

```java
// **Effects**: makes a new Poly = 0
public init()

// **Requires**: n >= 0
// **Effects**: makes a new Poly = c * x^n
public init(c: Int, n: Int)
```

(Note: full specs omitted to save space; style might not be perfect either – focus on main ideas.)
**Poly: Observers**

```swift
/// The degree of self, ie largest exponent with a non-zero coefficient, or 0 if self = 0.
public var degree : Int

/**
 **Requires**: d >= 0
 - Returns: The coefficient of the term of self whose exponent is d.
 */
public func coefficient(for d: Int) -> Int
```

**Poly: Producers**

```swift
/// - Returns: self + q, as a Poly
public func add(_ q : Poly) -> Poly

/// - Returns: self * q, as a Poly
public func mul(_ q : Poly) -> Poly

/// - Returns: -self
public func negate() -> Poly
```

```swift
let p = Poly(2,4)
let q = p.mul(p)
let r = q.negate()
```

---

**Aside: Operator Overloading**

```swift
/// - Returns: p + q
static public func +( _ p : Poly, _ q : Poly) -> Poly

/// - Returns: p * q
static public func *(_ p : Poly, _ q : Poly) -> Poly

/// - Returns: -p
static public prefix func -( _ p : Poly) -> Poly
```

```swift
let p = Poly(2,4)
let q = p * p
let r = -q
```

---

**IntSet: Overview, Abs State, Creator**

```swift
/// Overview: An IntSet is a mutable, unbounded set of integers. A typical IntSet is { x1, ..., xn }.
class IntSet {
    // **Effects**: makes a new IntSet = {}
    public init()
```
IntSet: Observers

/// - Returns: true if and only if element in self
public func contains(_ element: Int) -> Bool

/// Number of elements in the set
public var count : Int

/// - Returns: Some element of self.
/// - Throws: EmptyError if self is empty
public func choose() throws -> Int

IntSet: Mutators

/// **Modifies**: self
/// **Effects**: self_post = self_pre U { element }
public func add(_ element : Int)

/// **Modifies**: self
/// **Effects**: self_post = self_pre - { element }
public func remove(_ element : Int)