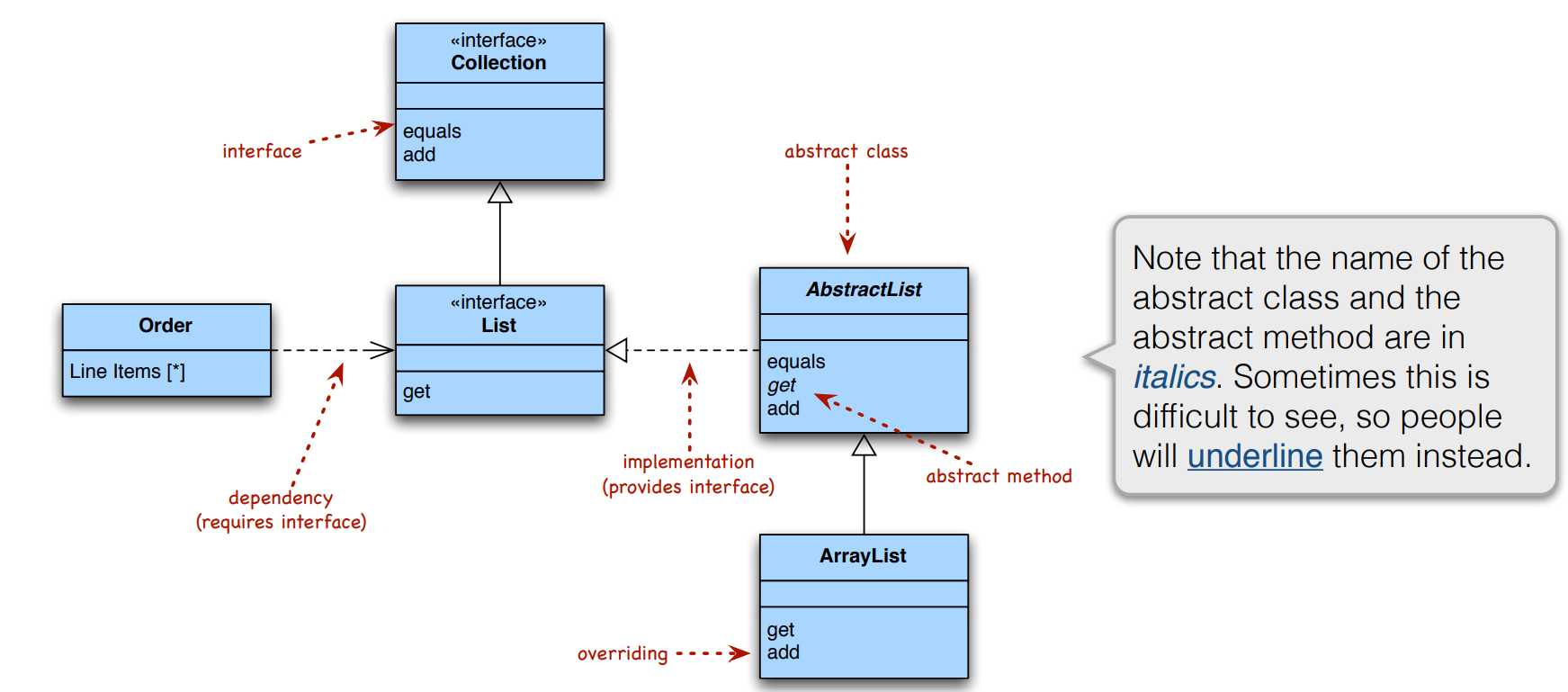
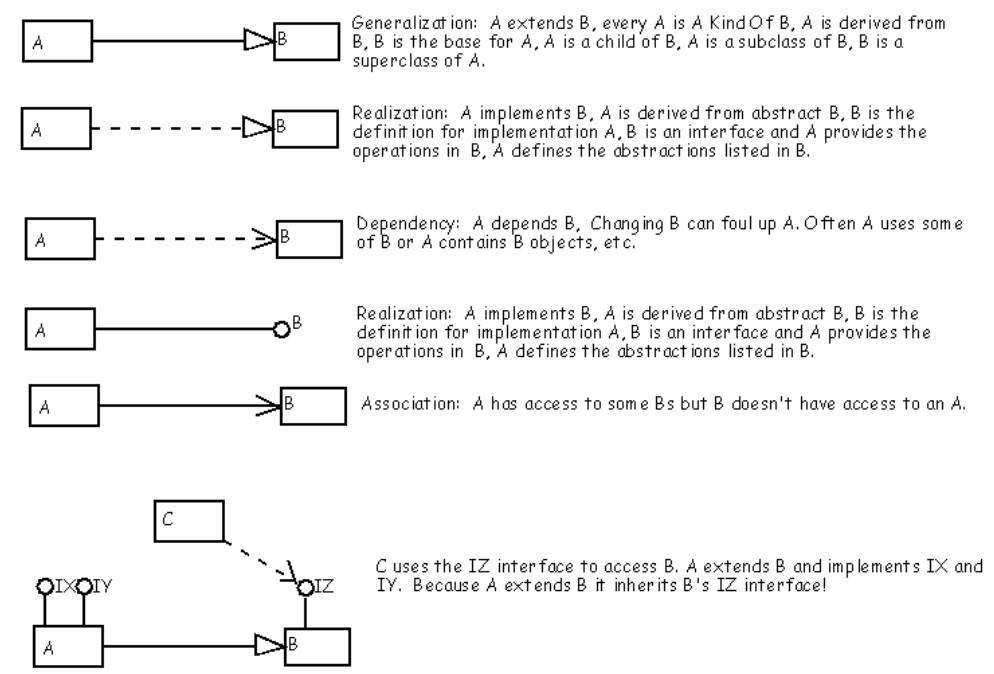
**UML Class Diagrams**

* 
* 

**Functional Programming**

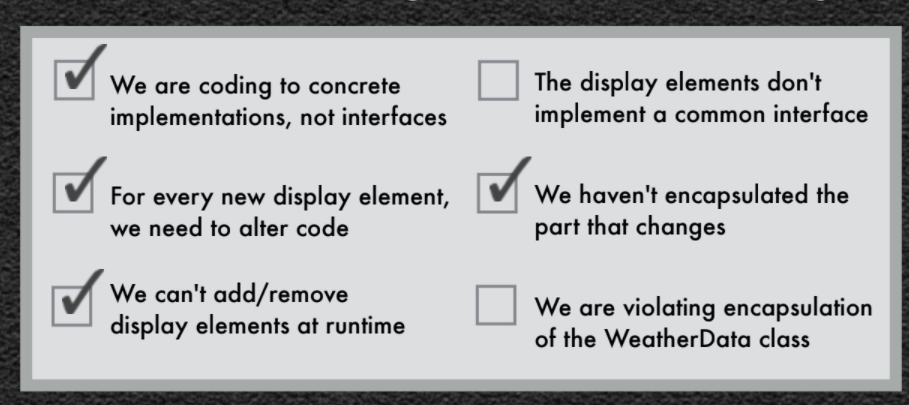
* pure functional language does not allow mutable objects True
* A pure functional language does not allow re-assignment to variables True
* A pure functional language does not allow iteration (while/for loops) True
* A pure functional language does not allow functions as parameters False?
* All (functional) list operations can be implemented using recursion and the base list

operations isEmpty, head, and tail True?

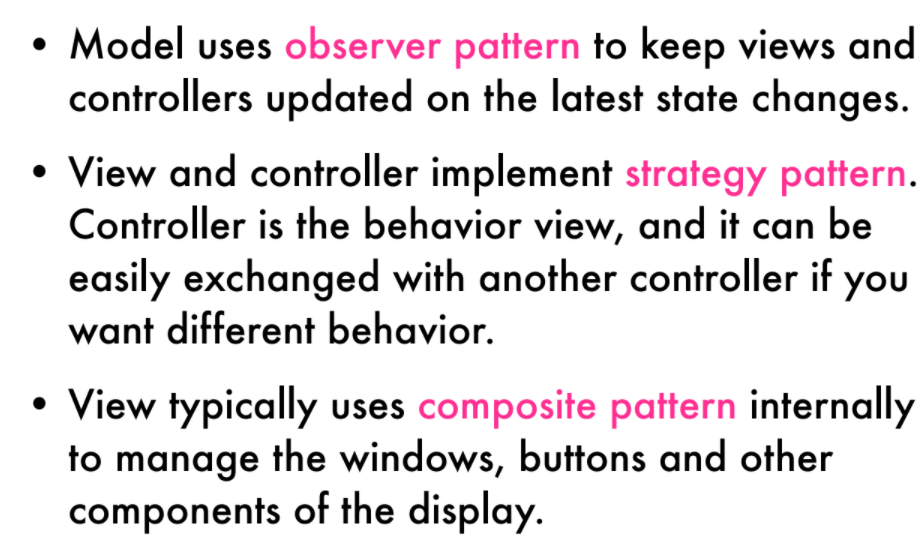
* Prepending an element to the beginning of a list takes constant time True
* Appending an element to the end of a list takes constant time False

**HFDP**

* OO design principles:
  + Encapsulate what varies: Identify the aspects of your application that vary and separate them from what stays the same
  + Favor composition over inheritance (Prefer a HAS-A relationship to an IS-A relationship)
  + Program to interfaces, not implementations
  + Strive for loosely coupled designs between objects that interact
  + Open-Closed Principle: Classes should be open for extension but closed for modification
  + Dependency Inversion Principle: Depend on abstractions; do not depend on concrete classes
  + Single Responsibility: A class should only have one reason to change
  + Favor composition over inheritance: Allows you to encapsulate a family of classes and change behavior at runtime
    - Has-A rather than Is-A
    - More flexibility – can change behavior easily at runtime
    - Reasoning is simplified and localized
  + The **Strategy Pattern** defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.



* + The **Observer Pattern** defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are *notified* and updated automatically. **Strive for loosely coupled designs between objects that interact**.
  + When two objects are loosely coupled, they can interact, but have very little knowledge of each other
  + Loosely coupled designs can handle change easier because they minimize the interdependency between objects
  + Classes should be open for extension, but closed for modification.
    - Open – Feel free to extend our classes with new behavior if your needs and requirements change (as they will).
    - Closed – We spent a lot of time getting this code correct and bug free, so we can’t let you alter existing code.
  + The **Decorator Pattern** attaches additional functionality to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
  + The **Simple Factory** is a programming idiom rather than a design pattern. An **idiom** is an expression of a simple task that may not be built-in to the programming language being used.
  + Depend upon abstractions. Do not depend upon concrete classes.
  + The **Abstract Factory Pattern** provides an interface for creating families of related or dependent objects without specifying their concrete classes.
  + The **Singleton Pattern** ensures a class has only one instance, and provides a global point of access to it. (Many designers would call this an ***idiom***)
  + The **Adapter Pattern** converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.
  + The **Iterator Pattern** provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. (Some designers might call this an ***idiom*** also)
  + A class should have only one reason to change
  + A class has **high cohesion** when it is designed around a set of related functions.
  + Every responsibility of a class is an area of potential change. More than one responsibility means more than one area of change.
  + The **Composite Pattern** allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
  + Model uses observer pattern
  + View and controller implement strategy pattern.
  + View typically uses composite pattern



* Observer: The model has no dependencies on viewers or controllers!
* Strategy: The view only worries about presentation, the controller worries about translating user input to actions on the model.
* Composite: The view is a composite of GUI components (labels, buttons, text entry, etc.). The top level component contains other components, which contain other components and so on until you get to the leaf nodes.
* Scala:
  + Uses the JVM
  + Object-oriented
  + Functional
* Functional:
  + Immutability (can’t change an object once it has been declared)
  + Variable not re-assignable (all variables are constants)
  + Uses recursion instead of loops (iteration)
  + Treat functions as first-class object (pass them as parameters; declare them)
  + Lists are typical data structure (as opposed to arrays)
* "This"::"is"::"a" :: Nil
* x (object i.e. String) is the head, xs is the tail (List)

**Design of Everyday Things**

* Major Areas of Design:
  + Industrial Design – Optimize function, value, and appearance of products and systems for the mutual benefit of both user and manufacturer (Form and Material)
  + Interaction Design – Focus on how people interact with technology. Goal is to enhance people’s understanding of what can be done, what is happening, and what has just occurred (Understandability and Usability)
  + Experience Design – Focus on quality and enjoyment of the total experience. (Emotional Impact)
* Fundamental Principles of Design:
  + Discoverability – It is possible to determine what actions are possible and the current state of the device.
  + Feedback – There is full and continuous information about the results of actions and the current state of the product or service. After an action has been executed, it is easy to determine the new state.
  + Conceptual model – The design projects all the information needed to create a good conceptual model of the system, leading to understanding and a feeling of control. The conceptual model enhances both discoverability and evaluation of results.
  + Affordances – The proper affordances exist to make the desired actions possible.
  + Signifiers – Effective use of signifiers ensures discoverability and that the feedback is well communicated and intelligible.
  + Mappings – The relationship between controls and their actions follows the principles of good mapping, enhanced as much as possible through spatial layout and temporal contiguity.
  + Constraints – Providing physical, logical, semantic, and cultural constraints guides actions and eases interpretation.
* When people encounter a device, they face two gulfs: the **Gulf of Execution**, where they try to figure out how to use it, and the **Gulf of Evaluation**, where they try to figure out what state it is in and whether their actions got them to their goal.
* Seven Stages of Action:
  + Goal (form the goal) – What do I want to accomplish?
  + Plan (the action) – What are the alternative action sequences?
  + Specify (an action sequence) – What action can I do now?
  + Perform (the action sequence) – How do I do it?
  + Perceive (the state of the world) – What happened?
  + Interpret (the perception) – What does it mean?
  + Compare (the outcome with the goal) – Is this okay? (Have I accomplished my goal?)
* The **designer’s conceptual model** is the designer’s conception of the look, feel, and operation of a product.
* The **system image** is what can be derived from the physical structure that has been built (including documentation).
* The **user’s mental model** is developed through interaction with the product and the system image.
* Designers expect the user’s model to be identical to their own, but because they cannot communicate directly with the user, the burden of communication is with the system image.
* An **affordance** is a property of an object that allows an individual to perform an action
  + Instead of allowing an action, an **anti-affordance** prevents it.
  + A **false affordance** is an apparent affordance that does not have any real function. The actor perceives nonexistent possibilities.
  + A **hidden affordance** indicates that there are possibilities for action, but these are not perceived by the actor.
  + A **perceptible affordance** is one where information is available so the actor perceives and can act upon the affordance.
* A **signifier** is any mark or sound or other perceivable indicator that communicates appropriate behavior to a person
  + tells people what to do and where to do it
  + A perceptible affordance may be a signifier, but a signifier does not have to be an affordance. A sign might tell a person what to do, but it does not determine if the action is possible
* Mapping is an important concept in the design and layout of controls and displays
* Constraints:
  + Physical – Physically prevent a wrong action from succeeding (should be apparent)
  + Cultural – Rely on accepted cultural conventions
  + Semantic – Rely on the meaning of the situation
  + Logical – Spacial or functional layout of components makes something obvious

1. Give the three OO design principles that the Strategy Pattern uses according to HFDP.
   1. Encapsulate what varies
   2. Favor composition over inheritance
   3. Program to interfaces, not implementations
2. Give the three OO design principles that the Observer Pattern uses according to HFDP.
   1. Strive for loosely coupled designs between objects that interact
   2. Encapsulate what varies
   3. Program to interfaces, not implementations
3. What are observers called in the Swing framework? (see “Other places you’ll find the Observer Pattern in the JDK” in Chapter 2) EventListeners
4. What is the main OO design principle introduced in the discussion of the Decorator Pattern?

Open-Closed

Adapter pattern is considered as a wrapper as well as Decorator

1. The Java I/O package is largely based on decorator. InputStream represents the abstract component. What class represents the corresponding abstract decorator? (see Decorating the java.io classes in chapter 3) FilterInputStream
2. What is the main OO design principle introduced in the discussion of the Factory Method Pattern?

Dependency Inversion Principle

1. ~~What OO design principle does the Factory Method Pattern ignore?~~ Ignore this question
2. What kind of class (introduced in Java 1.5) does “Effective Java” by Joshua Block recommend for creating singleton objects in Java? Enum (it has limitation bc of inheritance that locks)
3. What keyword (instead of class) does Scala to create a singleton object? Object (declare it as object)
4. There are two forms of the Adapter Pattern: object and class adapters. Which one did we cover? (see bullet points for chapter 7) Class adapter
5. What is the main OO design principle introduced in the discussion of the Iterator Pattern? Single Responsibility
6. What OO design principle does the Composite Pattern violate? (see the section just before “Flashback to Iterator” in chapter 9) Single Responsibility
7. The Java Swing components rely heavily on the Composite Pattern. Consider the following four JComponents: JPanel, JButton, JLabel, JMenu. Which are typically used as leaf classes and which are typically used as composite classes? Composite: JPanel, JMenu - Leaf: JButton, JLabel
8. Look at the Builder Pattern in HFDP. Give its brief description; tell whether it is a creational, structural, or behavioral pattern; and tell whether it is a class or object pattern.

Creational, Object Pattern

1. What is the name of the Builder for the String class? What is the name of its primary method? Is the builder class mutable or immutable?

StringBuilder, **append**/insert, mutable

* Single Responsibility -> Cohesion
* Loosely Coupled -> Coupling