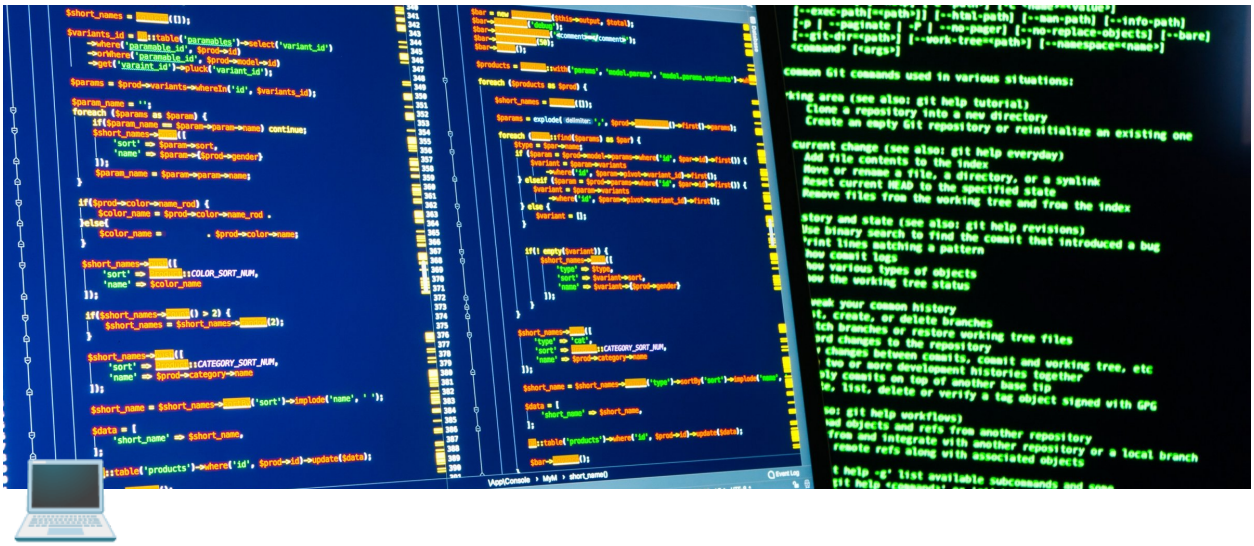


CSCB07 Notes

Lec Notes

TOPIC: SDLC and Version Control

Software Development Life Cycle:

1. identifying the problem (getting prompt/problem from client)
2. planning development stage
3. planning development stage (getting requirements from client; contact user w/ interviews)
4. design phase (this phase is where we design our software, what platform/languages to use, etc.)
5. implementation phase (translating design into code; ie. doing the actual coding/scripting/programming)
6. testing phase (testers in agile team will be doing the full testing: ways of testing include integrated testing, unit testing, etc.)
7. maintenance phase (the maintenance phase is where the developers will be there to maintain the software providing version updates (both minor and major)

Agile Software Development:

- scrum (certain way of building software in iterations on the go)

Version Control:

- systems: svn and git (GitHub is web service based on git)
- svn is centralized, git is distributed
- repo (repository) is like a history of the project
- **Centralized:**
  - given a v1, v2, and a delta, it stores the original version ie.v1 and the delta changes which saves a lot of memory. The following versions v2,v3, … can be accessed by retrieving the original version plus all the changes.
  - checkout: getting a version of the project out of the repo to your local machine
  - main disadvantage: single point of failure, if v1 deleted, then everything gone
  - storing deltas is computationally expensive
- **Distributed:**
  - each developer (local machine) has a copy of the repo (local repos)
  - first get the original repo (clone)
  - essentially the centralized dynamic is done locally
  - checking out and committing to the repo on their local machine
  - then they would push their repo to the original (commonly shared) repo with their changes
  - once someone else pushes their repo with changes, we can pull their changes from the original repo to our local repo
  - how git works: if we have three files and only file 3 is changed then we store the new file 3 with the changes and links to the other two unchanged files.

TOPIC: OOP

Abstract Classes

What are abstract classes:

- an abstract class is a class that cannot be instantiated and serves as a blueprint for derived classes
- it can contain both abstract and non-abstract methods
- it can have instance variables and constructors
- it may provide default implementations for some or all of its methods

- it can enforce or provide common behavior for its derived classes
- a class can inherit from only one abstract class

What can it be used for:

- when you want to provide a common base implementation for a group of related classes
- when you have some methods that can be fully implemented in the base class and other methods that should be implemented in the derived classes
- when you want to define common state or behavior that derived classes can inherit

Interface

What are interfaces:

- an interface is a contract that defines a set of methods that a class must implement
- it cannot have instance variables or constructors
- all methods in an interface are implicitly abstract and public
- it cannot provide default implementations for methods
- a class can implement multiple interfaces

What can it be used for:

- when you want to define a contract that specifies a set of behaviors that a class must implement.
- when you want to achieve multiple inheritances since a class can implement multiple interfaces
    - when you want to provide a common behavior across unrelated classes
- when you want to enable loose coupling by programming to interfaces rather than concrete classes

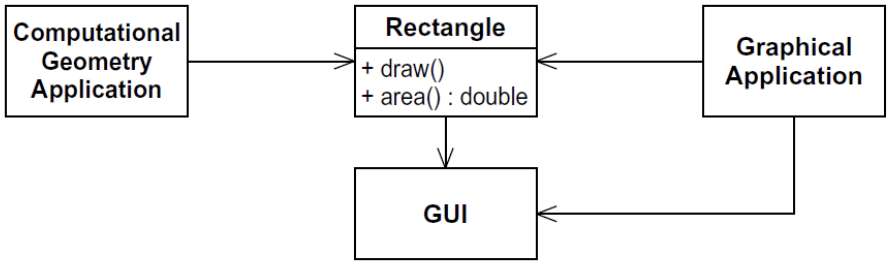
---

TOPIC: Solid Design

Single Responsibility Principle

- *a class should have only one reason to change*
- The following is an example of violating the SRP:

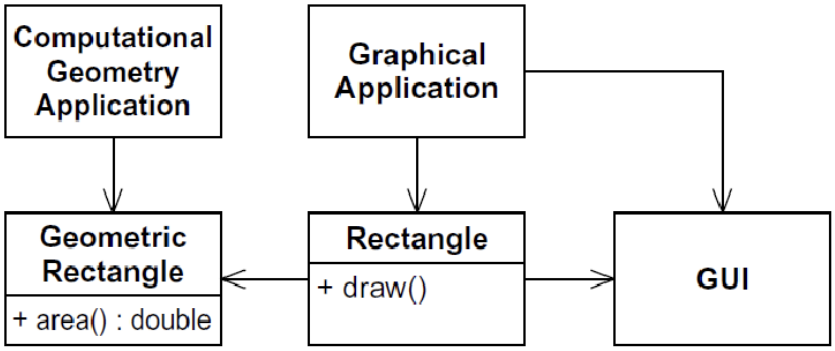
Violating the SRP (example)



Example: handling graphics of rectangle is one responsibility, calculating geometry of rectangle is another responsibility; the designer put both together which violates SRP. Note: change to GUI can affect computing of rectangle because the code is stored in the same Rectangle Class.

- The following is an example of conforming to the SRP:

Conforming to the SRP (example)



Here we see separate classes where computational geometry application takes care of the mechanical stuff, whereas graphical application will use both, computational app is not connected with GUI. Import IDEA: “coupling”.

Open/Closed Principle

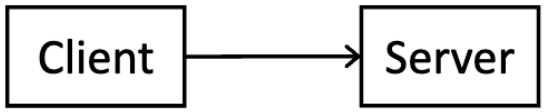
- probably the most important principle of SOLID
- software entities (classes, modules, functions, etc.) should be *open for extension, but closed for modification.*
- avoid modifying existing classes b/c:

- it may introduce more/new bug
- dependent classes can be affected

- The following is an example of a violation of OCP

Violating the OCP (example)

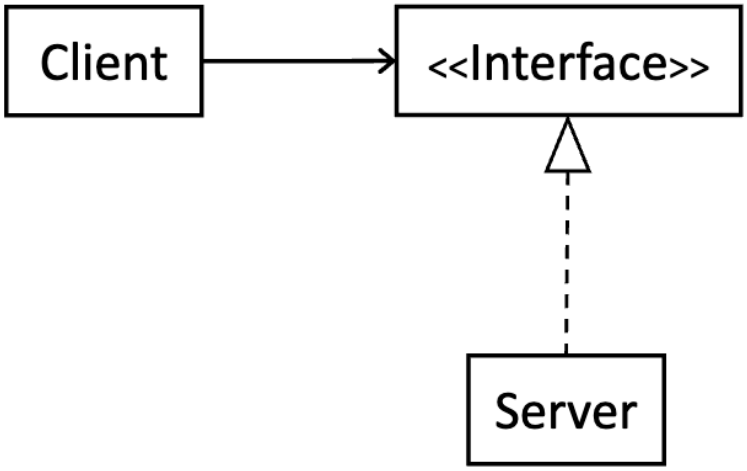
- Both classes are concrete
- The **Client** uses the **Server** class




```
package geometry

class Client{
  AdditionServer server;

  public void foo(int a, int b) {
    return server.compute(a,b);
  }
}

class AdditionServer{
  public void compute(int a, int b) {
    return a+b;
  }
}

public class OCPExample {
  public static void main(String[] args) {
    Client c = new Client(new ServerAdd));
    c.foo(2,3);
  }
}
```

- The following is an example of conforming to the OCP

Conforming to the OCP (example)



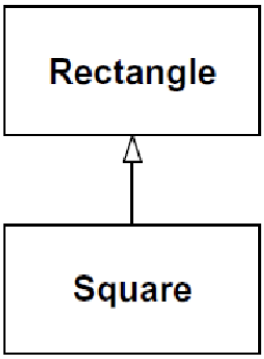
Liskov Substitution Principle

- *subtypes must be substitutable for their base types*
- Intuitive definition: idea that if design conforms to LSP then any property that is valid for an object belonging to a supertype, this property should be true for a object belonging to a subtype.
- The following example is a violation of LSP:

Issues

- Inheriting **height** and **width**
- Overriding **setHeight** and **setWidth**
- Conflicting assumptions. For example:

```
void testRectangleArea(Rectangle r){
        r.setWidth(5);
        r.setHeight(4);
        assertEquals(r.computeArea(), 20);
}
```



Here we see that when setWidth is ran, we have height AND width being 5, then both being 4 in following line. Hence, when we call the method computeArea() we have that area of the rectangle is calculated as a square with dimensions of 4x4. So, we have area of 16 which doesn't equal to 20, thus we get a false.

```
class Rectangle {
  double height;
```

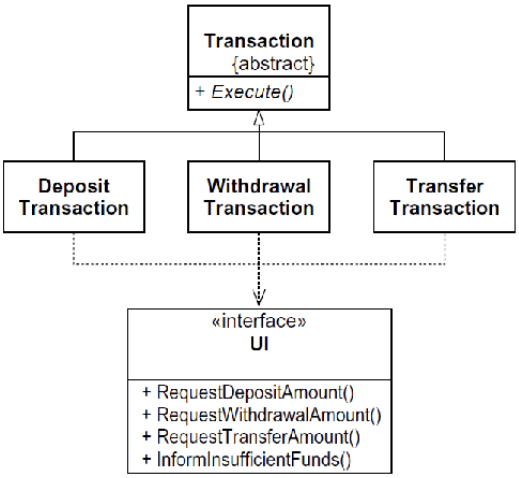
```
  double width;

  public Rectangle(double height, double width) {
    // ...
  }

  public void setWidth(double width) {
    this.width = width;
  }
  //...other methods
}

class Square extends Rectangle {
  public Square(double sideLength) {
    super(sideLength, sideLength);
    // makes the original width and height value of sideLength
  }

  // make sure we don't end with a square with dim. of (x,y) only (x,x)
  @Override
  public void setWidth(double width) {
    this.width = width;
    this.length = length;
  }

  // continue with same for length
  @Override
  public void setLength(double length) {
    this.width = width;
    this.length = length;
  }
}
```

- To fix the problem (example) from before we can have a superclass called Shape such that subclasses Rectangle and Square inherits the superclass.
- should not be making assumptions that clients/users will be be thinking the same way you are thinking

Interface Segregation Principle

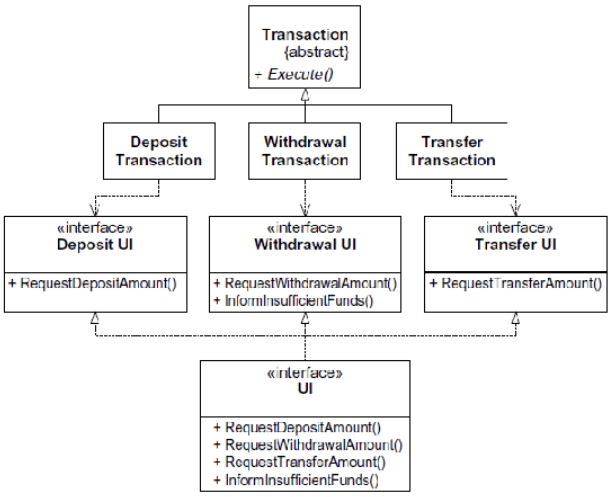
- *clients should not be forced to depends on methods that they do not use.*
- somehow similar to SRP
- interfaces should be coherent not cohesive
- so, interfaces of the class can be broken up into groups of methods where each group serves a different set of clients.
- The following is an example of ISP violation:

Violating the ISP (example)



- The following is an example of conforming to ISP:

Conforming to the ISP (example)



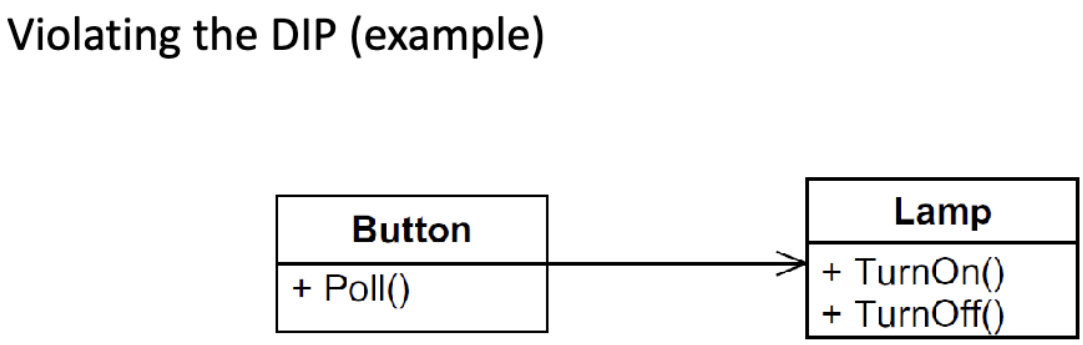
Here we want to split into three interfaces where each interface does its own part separately. So we see that for example Deposit Transaction will have a more coherent interface of Deposit UI, instead of having a cohesive interface of UI.

- NOTE: interfaces can inherit other interfaces (more than one is acceptable)

Dependency Inversion Principle

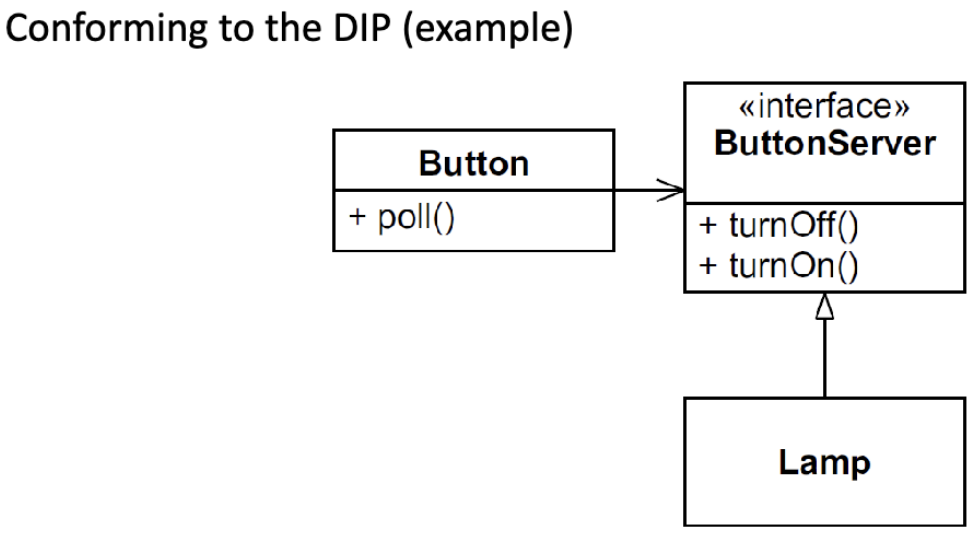
- A. High-level modules should not depend on low-level modules. Both should depend on abstractions.
- B. Abstraction should not depend on details. Details should depend on abstraction.

- NOTE: high-level classes basically sets policies, whereas the low-level classes is like a tool for the high-level classes.
- The following is an example of violation of DIP:



Here we see that Button is the high-level class and Lamp is the low-level class. So we are violating the DIP because Button depends on Lamp. So to fix this we want to have an interface of a button server.

- The following is an example of conforming to the DIP:



No reference to lamp, only to ButtonServer. Then anything that implements this interface will be able to use Button as they can access it through ButtonServer.

TOPIC: Design Smells

- symptoms of poor design

- often caused by violation of one or more SOLID Principles
- These symptoms include:
  - **Rigidity** - design is hard to change
  - **Fragility** - design is easy to break
  - **Immobility** - design is hard to reuse
  - **Viscosity** - is hard to do the right thing (if someone wants to follow your design and make changes based on that design, this would be more challenging than doing a hack)
  - **Needless Complexity** - overdesign; making code too complex
  - **Needless Repetition** - mouse abuse (ie. too much copy and paste)
  - **Opacity** - disorganized expression (when code becomes difficult to read and/or to understand)

  Note: refactoring can solve the design smell of opacity, documentation can also help

TOPIC: Design Patterns

- design patterns are solutions, not exact but generic; they are more specific than SOLID
- 23 design patterns divided into three categories:
  - Creational Patterns
  - Structural Patterns
  - Behavioral Patterns

Creational Patterns

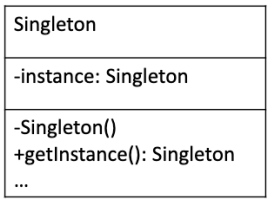
- concern the process of object creation
- 6 creational patterns:
  - factory method
    - intent: define an interface for creating an object, but let subclasses decide which class to instantiate
    - whenever creating objects with same or close related behavior
    - uses encapsulation
    - An example:

```
// Pizza interface and its implementations
interface Pizza {
```

```
    void prepare();
    void bake();
    void cut();
    void box();
}

class CheesePizza implements Pizza {
    @Override
    public void prepare() {
        System.out.println("Preparing Cheese Pizza");
    }

    @Override
    public void bake() {
        System.out.println("Baking Cheese Pizza");
    }

    @Override
    public void cut() {
        System.out.println("Cutting Cheese Pizza");
    }

    @Override
    public void box() {
        System.out.println("Boxing Cheese Pizza");
    }
}

class PepperoniPizza implements Pizza {
    // Similar methods as CheesePizza, but with Pepperoni-specific steps
}

class VeggiePizza implements Pizza {
    // Similar methods as CheesePizza, but with Veggie-specific steps
}

// PizzaFactory interface with its implementations
interface PizzaFactory {
    Pizza createPizza();
}

class CheesePizzaFactory implements PizzaFactory {
    @Override
    public Pizza createPizza() {
        return new CheesePizza();
    }
}

class PepperoniPizzaFactory implements PizzaFactory {
    // Similar implementation as CheesePizzaFactory, but for PepperoniPizza
}

class VeggiePizzaFactory implements PizzaFactory {
    // Similar implementation as CheesePizzaFactory, but for VeggiePizza
}

// Client code
```

```
public class Main {
    public static void main(String[] args) {
        PizzaFactory factory = new CheesePizzaFactory();
        Pizza pizza = factory.createPizza();

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
    }
}
```

- abstract factory
- singleton
  - intent: ensure a class has only one instance, and provide a global point of access to it



  - Example of Singleton:

```
// final class to prevent other classes from extending it
final class Manager {
  private static Manager manager;

  // private constructer to prevent instantiation outside the class
  private Manager() {
  }

  public static Manager getInstance() {
    if (manager == null) {
      manager = new Manager();
    }
    return manager;
  }
}

public class SingletonExample {
  public static void main(String [] args) {
    Manager m1 = Manager.getInstance();
    Manager m2 = Manager.getInstance();
    System.out.println(m1 == m2);
    // returns true, b/c pointing to same manager as there is only one manager b/c private
```

```
  }
}
```

- prototype
- builder
- object pool

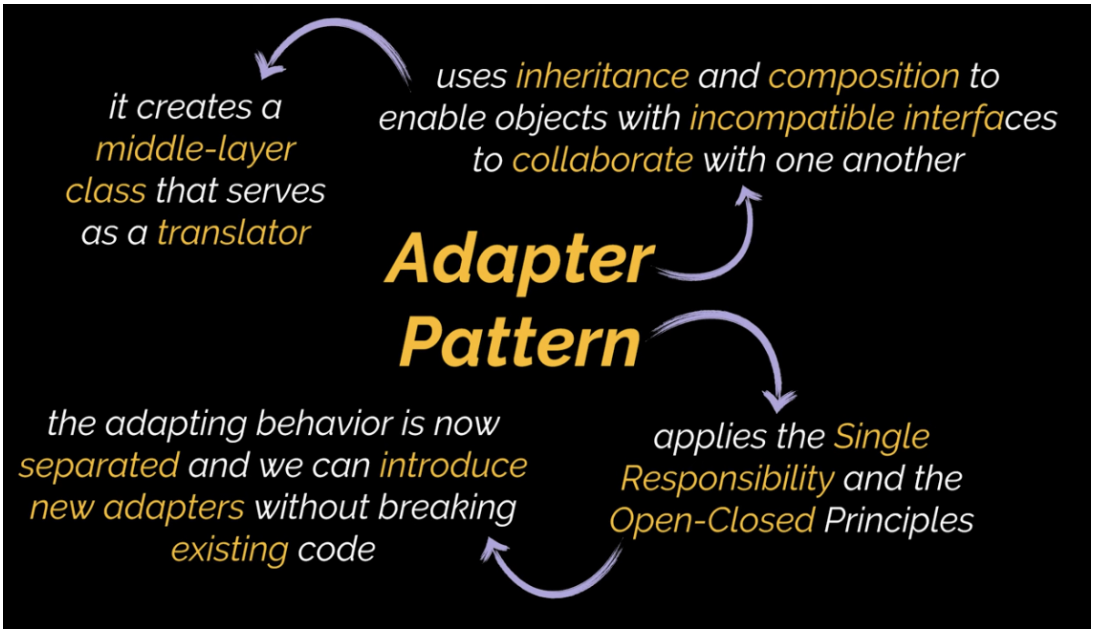
Structural Patterns

- deal with composition of classes or objects
- 7 structural patterns:
  - adapter
    - very important
    - intent: let classes work together that couldn't otherwise because of incompatible interfaces
    - helps things work together
    - note: for each adaptee there should be a corresponding adapter
    - Example of Adapter design pattern:

```
class Client {
  public void foo(Target t) {
    System.out.println(t.add(2,3));
  }
}

interface Target {
  public int add(int a, int b);
}

class Adaptee {
  public int sum(int a, int b) {
    return a + b;
  }
}

class Adapter implements Target {
  Adaptee adaptee;

  public Adapter(Adaptee adaptee) {
    this.adaptee = adaptee;
  }

  public int add(int a, int b) {
    return adaptee.sum(a,b);
```

```
  }
}
```

- More self notes:
  - allows objects with incompatible interfaces to collaborate with one another



```
// Existing interface for working with XML-based weather data
interface XMLWeatherService {
    void fetchXMLWeatherData();
}

// Existing class implementing the XMLWeatherService interface
class XMLWeatherServiceImpl implements XMLWeatherService {
    @Override
    public void fetchXMLWeatherData() {
        System.out.println("Fetching weather data in XML format...");
        // Code to fetch XML weather data
    }
}

// New interface for working with JSON-based weather data
interface JSONWeatherService {
    void fetchJSONWeatherData();
}

// New weather service providing data in JSON format
class JSONWeatherServiceImpl implements JSONWeatherService {
    @Override
```

```
    public void fetchJSONWeatherData() {
        System.out.println("Fetching weather data in JSON format...");
        // Code to fetch JSON weather data
    }
}

// Adapter class to make JSONWeatherService compatible with XMLWeatherService
class JSONToXMLAdapter implements XMLWeatherService {
    private JSONWeatherService jsonWeatherService;

    public JSONToXMLAdapter(JSONWeatherService jsonWeatherService) {
        this.jsonWeatherService = jsonWeatherService;
    }

    @Override
    public void fetchXMLWeatherData() {
        jsonWeatherService.fetchJSONWeatherData();
        // Additional code to convert JSON data to XML format if needed
        System.out.println("Converting JSON data to XML...");
    }
}

// Client code
public class Main {
    public static void main(String[] args) {
        // Using the JSONWeatherService directly
        JSONWeatherService jsonWeatherService = new JSONWeatherServiceImpl();
        jsonWeatherService.fetchJSONWeatherData();

        // Using the XMLWeatherService through the adapter
        XMLWeatherService xmlWeatherService = new JSONToXMLAdapter(jsonWeatherService);
        xmlWeatherService.fetchXMLWeatherData();
    }
}
```

- bridge
- composite
- decorator
- facade
  - intent: hide complexities and provide a unified interface to a set of interfaces in a subsystem
  - instead of requiring client to invoke a lot methods, create a simple function that takes input from user then tackle the complexity and return output to user.
  - help with client experience (they don’t need to go through all the complex system)
  - facade is written by people who are managing/developing the complex system, not the client.

- Example:

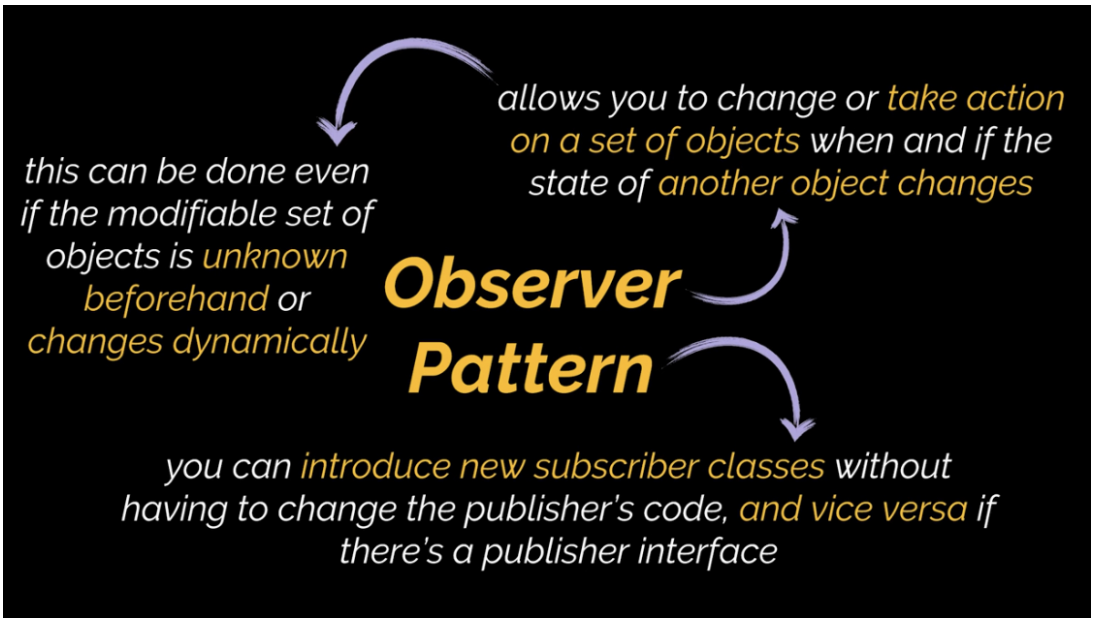
```
// Complex subsystems
class Graphics {
    void initialize() {
        // Initialize graphics
    }

    void render() {
        // Render graphics
    }
}

class Audio {
    void initialize() {
        // Initialize audio
    }

    void playSound() {
        // Play sound
    }
}

class Physics {
    void initialize() {
        // Initialize physics
    }

    void updatePhysics() {
        // Update physics calculations
    }
}

// Game Engine facade
class GameEngine {
    private Graphics graphics;
    private Audio audio;
    private Physics physics;

    public GameEngine() {
        graphics = new Graphics();
        audio = new Audio();
        physics = new Physics();
    }

    void initialize() {
        graphics.initialize();
        audio.initialize();
        physics.initialize();
    }

    void runGameLoop() {
        while (true) {
            graphics.render();
            audio.playSound();
            physics.updatePhysics();
```

```
            // Other game-related logic
        }
    }
}

// Main game class
public class Main {
    public static void main(String[] args) {
        GameEngine gameEngine = new GameEngine();
        gameEngine.initialize();
        gameEngine.runGameLoop();
    }
}
```

- flyweight
- proxy

Behavioral Patterns

- characterize the ways which classes or objects interact and distribute responsibility
- 10 behavioral patterns:
  - chain of responsibility
  - command
  - interpreter
  - iterator
  - mediator
  - memento
  - observer
    - intent: define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
    - Self Notes:
      - an example is like a subscription system for store updates
      - observer dp notifies multiple objects, or subscribers, about any events that happen to the object they’re observing, or publisher
      - makes code more open for extension (OCP)



- Example of observer design pattern:

```
public class Store {
  private final NotificationService notificationService;

  public Store() {
    notificationService = new NotificationService();
  }

  public void newItemPromotion() {
    notificationService.notify();
  }

  public NotificationService getService() {
    return notificationService;
  }
}

public class NotificationService {
  private final List<EmailMSGListener> customers;

  public NotificationService() {
    customers = new ArrayList<>();
  }

  public void subscribe(EmailMSGListener listener) {
    customers.add(listener);
  }

  public void unsubscribe(EmailMSGListener listener) {
    customers.remove(listener);
  }

  public void notify() {
```

```
    customers.forEach(listener -> listener.update());
  }
}

public class EmailMSGListener implements EventListener{
  private final String email;

  public EmailMSGListener(String email) {
    this.email = email;
  }

  @Override
  public void update() {
    // Actually send the email
  }
}

public class MobileAppListener implements EventListener {
  private final String username;

  public MobileAppListener(String username) {
    this.username = username;
  }

  @Override
  public void update() {
    // Actually send the push notification
  }
}

public interface EventListener {
  void update();
}

// Client code
public static void main(String [] args) {
  Store store = new Store();
  store.getNotificationService().subscribe(
    new EmailMSGListener("geekific@like.com")
  );
  store.getNotificationService().subscribe(
    new EmailMSGListener("geekific@subs.com")
  );
  store.getNotificationService().subscribe(
    new MobileAppListener("GeekificLnS")
  );
  store.newItemPromotion();
}
```

- state
- strategy

        - intent: define a family of algorithms, encapsulate each one, and make them interchangeable
        - allows for conforming to OCP and SRP
        - define a family of algorithms, puts each of them in a separate class, and makes their objects interchangeable
        - strategy class has no visibility on how “payment” is being conducted as it is making use of strategy interface (PaymentStrategy)
        - can add new algorithms or change existing ones without changing code of service or other strategies.
        - define strategy and pass to service
        - example:
            - think of ConcreteStrategy (add, subtract, multiply, divide)
            - client wants to access a class Content where we are trying to determine what math operation we want to use.
            - so we can apply a strategy
            - create an interface Strategy that is used by Context to execute a strategy (choose an operation).
            - so from Context, the client can access an operation whilst the devs can add new operations such as exponents, etc. and also modify operations is needed.
    - template

CSCB07 Midterm Review Session

SDLC

- Planning - develop a plan for creating the concept or evolution of the concept
- Analysis - analyze the needs of those using the system. Create detailed requirements
- Design - coming up with high level descriptions of system to be implemented in implementation phase (uml)
- Implementation - where you complete the work of developing and testing the system

- Maintenance - doing any needed maintenance to keep system running (fix bugs, optimize performance, accommodate new tech)

Waterfall

- a sequential (non-iterative) model
- involves a large amount of upfront work, in a n attempt to reduce the amount of work done in later phases of the project
- main disadvantage: need to finalize a certain phase before starting the next phase (complete stages perfectly); no flexibility
- advantages: for small projects, easy to implement (structured)

Spiral

- many iterations of Waterfall
- risk-driven model
- more time is spent on a given phase based on the amount of risk that phase poses for the project
- used for high risk projects (projects that requires a lot of money; healthcare, finance, nuclear reactor)
- disadvantage: require people who a very specific skill set; high costs

Agile

- issues with Waterfall
  - inappropriate when requirements change frequently
  - time gets squeezed the further into the process you get
- **Agile Methodologies**
  - extreme programming (XP)
  - scrum
  - test-driven development (TDD)
  - feature-driven development (FDD)
- Agile Manifesto
  - Individuals and interactions over processes and tools
  - working software over comprehensive negotiation

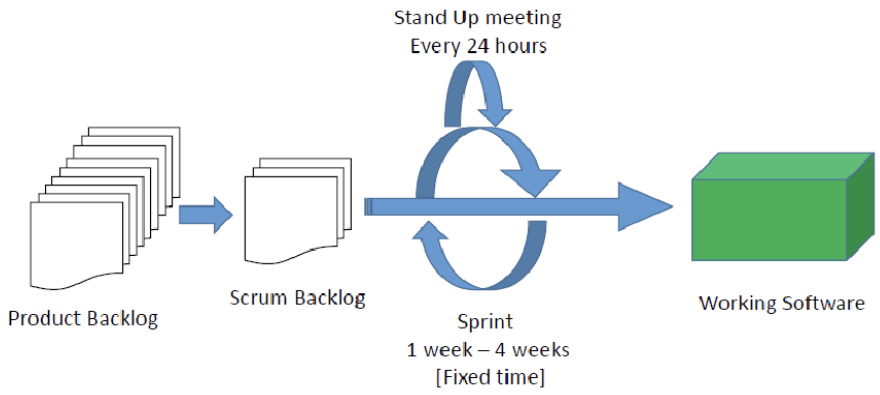
- customer collaboration over contract negotiation
- responding to change over following a plan (major principle)

eXtreme Programming (XP)

- one of the most rigorous forms of Agile
- one person codes while others watch them
- requires extreme amounts of testing

Scrum

- scrum is currently one of the most widely used methodologies of software development
- Scrum process:



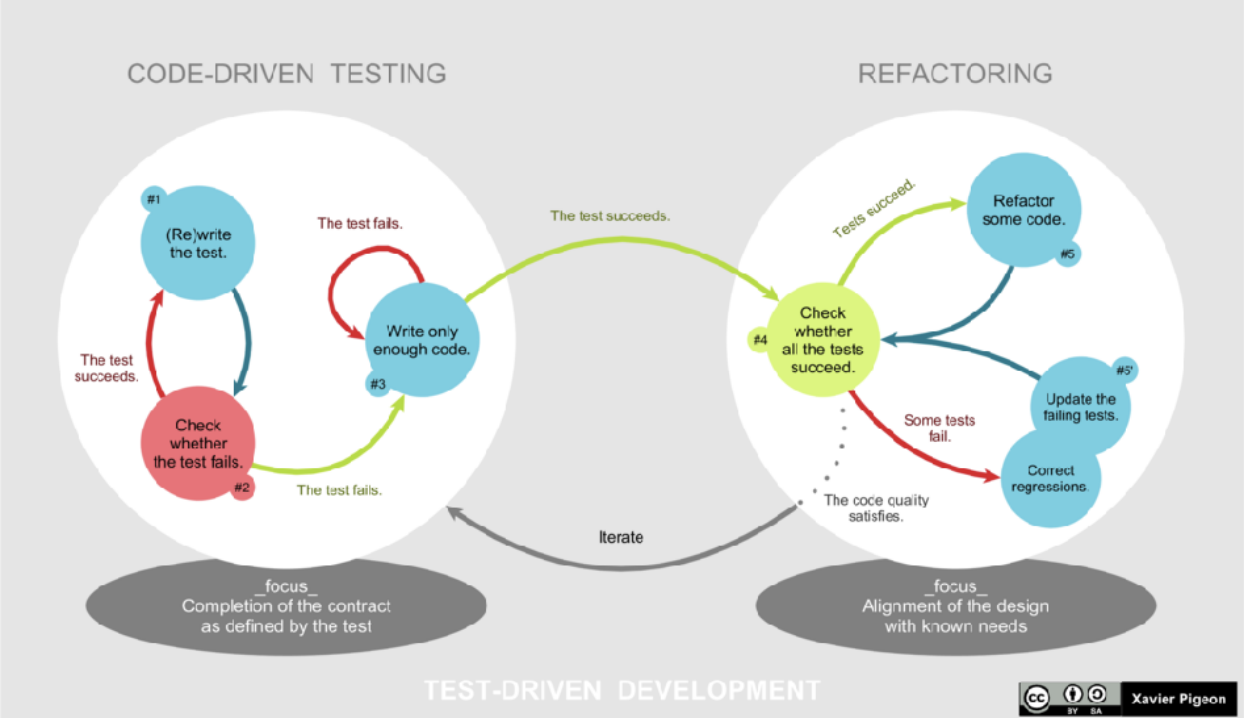
Note: backlog is dynamic

- ROLES:
  - Product Owner
    - responsible for delivering requirements and accepting demos
    - involved in planning session
    - needs to talk to stakeholders
    - person who decides what goes in product backlog and what goes in scrum backlog
  - Scrum Master
    - responsible for removing impediments

        - not allowed to tell people what to do
        - job is to facilitate things, make job easier for team members
    - Team Members
        - no one has a fixed role other than the scrum master and product owner
        - everyone takes on tasks, and completes them based on what they are most comfortable with
- SPRINT:
    - a fixed time to deliver a working set of features, that are reviewed in a demonstration to the product owner
    - broken into “User Stories”
        - User Story: “As a {ACTOR/OBJECT} I want to {ACTION} so that {RESULT}”
    - usually between 1-4 weeks in length
- PLANNING POKER:
    - In scrum, we do not assign time to tasks, but assign arbitrary points. This is a form of estimation that helps gauge how work something will take to complete.
- WORKING AGREEMENT:
    - a series of statements that everyone on the team agrees to about how the team will work
- DEFINITION OF DONE:
    - a formal agreement of when work is considered complete
    - it is important that team comes to an agreement on this definition before they start work

TEST DRIVEN DEVELOPMENT (TDD)

- TDD is a way to develop software that revolves around writing test cases
- The basic concept is to write the unit tests needed to be passed for a feature to be considered working. You then code to the unit tests –writing the minimum amount for the tests to succeed
- Once working, you review and refactor. Then move on to the next set of tests.



Note: write as less code as possible when correcting errors

FEATURE DRIVEN DEVELOPMENT (FDD)

- based on the idea of building a focused model for the project, and then iterating on the features needed
- splits development into 5 major pieces:
  - develop overall model
  - build feature list
  - plan by feature
  - design by feature
  - build by feature

Intro to Android

- VIEW:
  - most GUI components in Android applications are instances of the View class or one of its subclasses (ie. Button, EditText, ImageView)
- COMMON GUI COMPONENTS:

- TextView
- EditText
- Button
- Switch
- Spinner

STORING DATA - ANDROID

- Data storage options
  - file system
  - shared preferences
  - databases
    - SQLite, Firebase Realtime Database

File System

- android’s file system consists of six main partitions
  - /boot
  - /system
  - /recovery
  - /data
  - /cache
  - /misc
- reading/writing data to a file on internal storage can be done with:
  - openFileInput( )
  - openFileOuput( )

TESTING

- Example from lecture:

```
@Test
void test_gcd() {
  int ouput = Tools.gcd(4,6);
  assertEquals(output,2);
  // or assertTrue(output == 2);
}

class Tools {
  static Circle createCircle(...){
    /* ... */
  }

  /*
  ...
  */

  static int gcd(int a, int b){
    /* ... */
  }
}
```

- **Testing Tools:**
  - Testing (automated, manual testing)
    - most used nowadays
  - Reviews (code reviews)
  - Formal Verification
    - Disadvantages of formal verification:
      - proofs are hard
      - time consuming
      - not scalable (major disadvantage)

Testing Levels

- Acceptance testing
- System testing
- Integration testing
- Module testing
- Unit testing

Testing Approaches (Important concept)

- **Black-box testing**
  - test are derived from external descriptions of the software
  - testing without knowing internal structure of the box
  - easier and faster b/c don't need to know structure of method, only need to know input output of method.
  - thinking of the test in terms of the inputs and outputs of the method
  - don't know structure necessarily hence will be testing "blindly"
- **White-box testing**
  - test are derived from the source code internals of the software
  - more expensive to apply
  - actually have to know the structure of the method in order to the testing
  - I know what to test b/c of understanding of method structure (ie. can cover all edges and nodes)

Why is software testing hard?

- knowing when to stop (enough testing or not)
- random/statistical testing is not effective

Infamous Software Failures

- Northeast blackout of 2023
- Ariane 5 explosion (1996)
- NASA's Mars lander (1999)
- Therac-25 radiation therapy machine

Software Fault: A static defect in the software

Software Error: An incorrect internal state that is the manifestation of some fault

Software Failure: external, incorrect behavior with respect to the requirements or another description of the expected behaviour

RIPR (important to study)

- 4 conditions needed for failure to be observed
  - **Reachability:** a test must reach the location in the program that contains the fault
  - **Infection:** after the faulty location is executed, the state of the program must be incorrect
  - **Propagation:** the infected state must propagate through the rest of the execution and cause some output or final state of the program to be incorrect.
  - **Revealability:** the tester must observe part of the incorrect portion of the final program state

Control Flow Graph

- for each statement, draw these statements first
- have nodes as objects
- have operations as edges (links)
- node coverage doesn't imply edge coverage, but edge coverage implies node coverage

Logic Coverage

- involves the boolean expressions of the code
- coverage criteria:
  - predicate coverage
    - the test set should make each predicate evaluate to true and false
    - ie. ((a>b || c) && (x<y) = {True, False}
    - Consider: P = [(a>b) || c] && (x<y)

Test Cases	a>b	c	x<y	P
T1: a=5, b=4, c = T, x=3, y=2	T	T	F	F
T2: a=5, b=4, c=F, x=2, y=3	T	F	T	T

    - if want 100% clause coverage, then need to satisfy every clause (T,F) so in our example above, clause coverage is not satisfied (only predicate coverage). This is because (a>b) clause is only tested with T output.
    - NOTE: predicate coverage DOES NOT imply clause coverage, and clause coverage also DOES NOT imply predicate coverage.

▼ **EXAM Q**: Does clause coverage imply predicate coverage? If not give counter example.

No, here is a counter example.

	a	b	P
T1	T	F	F
T2	F	T	F

- clause coverage
  - the test set should make each clause evaluate to true and false
  - ie. (a>b) = {True, False}, c = {True, False}, (x<y) = {True, False}
  - weakness: the values do not always make a difference

QUESTION TO LOOK UP: find examples of cases where we should check active clause coverage additionally to clause coverage

- active clause coverage
  - stronger version of clause coverage (more powerful, covers more cases)
  - need to check whether there are two cases for each clause (true and determines the predicate & is false and determines the predicate)
  - ie. T $\rightarrow$ T, F $\rightarrow$ F for each clause (ie. a $\rightarrow$ P)
  - so overall, active clause coverage means that for each clause it should satisfy the properties of one clause (T) implies (P == T), and also vice versa.
- NOTE: active clause coverage implies clause coverage, **active clause coverage implies predicate coverage**

▼ (PROVE THIS)

Active Clause Coverage Implies Predicate Coverage

Proof:

1. Let's consider a single predicate P with n clauses: P = C1 AND C2 AND ... AND Cn.
2. Assume we have achieved active clause coverage for this predicate P. This means that each clause Ci within the predicate P has been the cause of P evaluating to both true and false at least once.``
3. Since we have achieved active clause coverage, we know that each individual clause Ci has been evaluated to both true and false in different test cases.

4. Now, let's analyze how the predicate P evaluates:
    - If all clauses Ci in P are true, then P is true.
    - If any one of the clauses Ci in P is false, then P is false.
5. Since we have achieved active clause coverage, we have tested all combinations of the clauses Ci evaluating to true and false within the predicate P.
6. This implies that we have tested all possible combinations of truth values for the predicate P, including cases where P is true and cases where P is false.
7. Therefore, by achieving active clause coverage for the predicate P, we have implicitly covered all possible combinations of truth values for the predicate P.
8. Since this proof holds true for any predicate P in the program, by achieving active clause coverage for all predicates, we have effectively covered all possible combinations of truth values for all predicates.
9. This aligns with the definition of predicate coverage: covering all possible combinations of truth values for the predicates in the program.

Active Clause Coverage Implies Clause Coverage (false)

Counterexample:

Consider a predicate P with two clauses: P = C1 AND C2.

For active clause coverage:

- Test case 1: C1 = true, C2 = true (P evaluates to true)
- Test case 2: C1 = false, C2 = false (P evaluates to false)

Active clause coverage is achieved because both C1 and C2 have caused P to evaluate to both true and false.

However, clause coverage is not achieved because we have not tested cases where each individual clause is evaluated to true and false separately:

- We haven't tested C1 in isolation with C2 = true (C1 = true, C2 = true).
- We haven't tested C2 in isolation with C1 = true (C1 = true, C2 = true).

Therefore, in this example, active clause coverage does not imply clause coverage.

Test Oracles

- test oracle is an encoding of the expected results of a given test

    - ie. JUnit assertion
- what should be checked:
    - output state is everything that is produced by the software under test
    - each test should have a goal and testers should check the output(s) that are mainly related to that goal
- how to determine what the correct results are:
- **specification-based direct verification of outputs**
    - produce a specific formula to test a method and match if output is correct or not
    - problem is that it is not scalable, need to write mathematical formula for each output which is very challenging and costly
- **redundant computations**
    - idea is get a program (software) that sorts arrays and feed same array to your program and the other well known program and see if outputs match
    - usually used for regression testing
- **consistency checks**
    - think of it as a sanity check
    - check whether certain properties hold

Android Testing

Model-View-Presenter (MVP)

- an architectural design pattern that results in code that is easier to test
- consists of 3 components:
    1. Model (Data)
    2. View (UI)
    3. Presenter (Business Logic)

Local and Instrumented Tests

- local unit tests
    - runs on the machine's local JVM
    - suggested around 70% unit tests

- instrumented tests
  - runs on the emulator or users actual device
  - often used for integrated testing or user interface testing
  - suggested around 20% integration test
  - suggested around 10% UI test

Commonly used tools

- JUnit
  - for unit tests in Java
- Mockito
  - used for creating mock objects to facilitate testing a component in isolation
- Espresso
  - used for writing UI tests

Mock Objects

- software component that is used to replace the “real” component during testing
- used for:
  - represent components that have not yet been implemented
  - speed up testing
  - reduce the cost
  - avoid unrecoverable actions

Mockito

- mocking framework for Java
- features include:
  - **creating mocks:** allows for creating mock objects and configure them
  - **stubbing:** process of specifying what to do if a specific method of the mock object is called
  - **verifying behavior:** check whether certain method has been invoked, etc.
- **Stubbing in Mockito**

    - A stub is a fake class that comes with preprogrammed return values. It's injected into the class under test to give you absolute control over what's being tested as input. A typical stub is a database connection that allows you to mimic any scenario without having a real database.

- **Mocking in Mockito**
    - mocking is the act of removing external dependencies from a unit test in order to create a controlled environment around it.
    - A mock is a fake class that can be examined after the test is finished for its interactions with the class under test. For example, you can ask it whether a method was called or how many times it was called. Typical mocks are classes with side effects that need to be examined, e.g. a class that sends emails or sends data to another external service.

▼ Why is the naive solution of pre-filling a realtime database with "customers" and running a test against it problematic?

- creates a hard dependency on a running database, and also requires an extra step to create the test data.
- not practical in real world

▼ What is the best solution to this?

The best solution for a true unit test is to completely remove the database dependency. We will stub the database connection instead, and "fool" our class to think that it is talking to a real "Manager", while in reality, the "Manager" is a Mockito stub.

https://semaphoreci.com/community/tutorials/stubbing-and-mocking-with-mockito-2-and-junit#:~:text=A stub is a fake,without having a real database.

- Example JUnit test case using Mockito in Java. Consider a simple 'Calculator' class with 'add' method.

```
// Calculator.java
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}

// CalculatorTest.java
import org.junit.Test;
import static org.mockito.Mockito.*;

public class CalculatorTest {

    @Test
    public void testAddition() {
```

```
        // Create a mock instance of Calculator
        Calculator calculatorMock = mock(Calculator.class);

        // Define the behavior of the mock
        when(calculatorMock.add(2, 3)).thenReturn(5);

        // Perform the actual test
        int result = calculatorMock.add(2, 3);

        // Verify the behavior
        verify(calculatorMock).add(2, 3);

        // Assert the result
        assertEquals(5, result);
    }
}
```