50.003 Elements of Software Construction
Final Exam Toolbox

Joel Huang

April 25, 2018

1 Software Design Patterns

1.1 Factory Design Pattern

Factory pattern is one of the most used design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

```
import java.util.Scanner;

public class PizzaStore {

  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    String whichPizza = "";
    System.out.print("What type of pizza? cheese/greek/pepperoni: ");

        if (sc.hasNextLine()){
            whichPizza = sc.nextLine();
        }

    PizzaFactory factory = new PizzaFactory();
    Pizza pizza = factory.makePizza(whichPizza);
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
  }
}

class PizzaFactory {

  public Pizza makePizza(String type) {
    Pizza pizza = null;
    String log = "";
    if (type.equals("cheese")) {
      pizza = new CheesePizza();
      log = "Made pizza of type " + pizza.getClass().getSimpleName();
    }
    else if (type.equals("greek")) {
      pizza = new GreekPizza();
      log = "Made pizza of type " + pizza.getClass().getSimpleName();
    }
    else if (type.equals("pepperoni")) {
      pizza = new PepperoniPizza();
      log = "Made pizza of type " + pizza.getClass().getSimpleName();
```

```
    } else {
      pizza = new CheesePizza();
      log = "We don't serve " + type +", instead we made you a " + pizza.getClass().
    getSimpleName();
    }
    System.out.println(log);
    return pizza;
  }
}

class Pizza {

  public void prepare() {
  }

  public void box() {
  }

  public void cut() {
  }

  public void bake() {
  }
}

class CheesePizza extends Pizza {}
class GreekPizza extends Pizza {}
class PepperoniPizza extends Pizza {}
```

1.2 Observer Design Pattern

Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically.

```
import java.util.ArrayList;
import java.util.List;

class Subject {
   private List<Observer> observers = new ArrayList<Observer>();
   private int state;

   public int getState() {
      return state;
   }

   public void setState(int state) {
      this.state = state;
      notifyAllObservers();
   }

   public void attach(Observer observer){
      observers.add(observer);
   }

   public void notifyAllObservers(){
      for (Observer observer : observers) {
         observer.update();
      }
   }
}

abstract class Observer {
   protected Subject subject;
   public abstract void update();
}


// two concrete implementations of the observer class

class BinaryObserver extends Observer{

   public BinaryObserver(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }

   @Override
   public void update() {
      System.out.println( "Binary String: " + Integer.toBinaryString( subject.getState() )
    );
   }
}

class OctalObserver extends Observer{

   public OctalObserver(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }

   @Override
   public void update() {
     System.out.println( "Octal String: " + Integer.toOctalString( subject.getState() ) );
   }
}
```

```
public class ObserverPattern {
   public static void main(String[] args) {
      Subject subject = new Subject();

      new OctalObserver(subject);
      new BinaryObserver(subject);

      System.out.println("First state change: 15");
      subject.setState(15);
      System.out.println("Second state change: 10");
      subject.setState(10);
   }
}

// output
// First state change: 15
// Octal String: 17
// Binary String: 1111
// Second state change: 10
// Octal String: 12
// Binary String: 1010
```

1.3 Decorator Design Pattern

Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class.

This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

```
import java.util.Scanner;

public class DecoratorDemo {
  public static void main(String[] args){
    MyPizza basicPizza = new Ham(new TomatoSauce(new Cheese(new PlainPizza())));
    System.out.println("Ingredients: " + basicPizza.ingredients());
    System.out.println("Price: " + basicPizza.getCost());
  }
}

interface MyPizza {
  public String ingredients();
  public double getCost();
}

class PlainPizza implements MyPizza {
  public String ingredients() {
    return "dough";
  }

  public double getCost() {
    return 4.00;
  }
}

abstract class ToppingDecorator implements MyPizza {
  protected MyPizza tempPizza;

  public ToppingDecorator(MyPizza newPizza){
    tempPizza = newPizza;
  }

  public String ingredients() {
    return tempPizza.ingredients();
  }

  public double getCost() {
    return tempPizza.getCost();
  }
}

class Cheese extends ToppingDecorator {

  public Cheese(MyPizza newPizza) {
    super(newPizza);
  }

  public String ingredients(){
    return tempPizza.ingredients() + ", cheese";
  }

  public double getCost(){
    return tempPizza.getCost() + 2.00;
  }
}
```

```
class TomatoSauce extends ToppingDecorator {

  public TomatoSauce(MyPizza newPizza) {
    super(newPizza);
  }

  public String ingredients(){
    return tempPizza.ingredients() + ", tomato sauce";
  }

  public double getCost(){
    return tempPizza.getCost() + .35;
  }
}

class Ham extends ToppingDecorator {

  public Ham(MyPizza newPizza) {
    super(newPizza);
  }

  public String ingredients(){
    return tempPizza.ingredients() + ", ham";
  }

  public double getCost(){
    return tempPizza.getCost() + 4.00;
  }
}
```

1.4 Strategy Design Pattern

In Strategy pattern, a class behavior or its algorithm can be changed at run time. This type of design pattern comes under behavior pattern.

In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object. The strategy object changes the executing algorithm of the context object.

```
interface Strategy {
   public int doOperation(int num1, int num2);
}

//Create concrete classes implementing the same interface.
class OperationAdd implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 + num2;
   }
}
class OperationSubstract implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 - num2;
   }
}
class OperationMultiply implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 * num2;
   }
}

// create a context
class Context {
   private Strategy strategy;

   public Context(Strategy strategy){
      this.strategy = strategy;
   }

   public int executeStrategy(int num1, int num2){
      return strategy.doOperation(num1, num2);
   }
}

// Use the Context to see change in behaviour when it changes its Strategy.
public class StrategyPattern {
   public static void main(String[] args) {
      Context context = new Context(new OperationAdd());
      System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationSubstract());
      System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationMultiply());
      System.out.println("10 * 5 = " + context.executeStrategy(10, 5));
   }
}

// output
// 10 + 5 = 15
// 10 - 5 = 5
// 10 * 5 = 50
```

1.5 Visitor Design Pattern

In Visitor pattern, we use a visitor class which changes the executing algorithm of an element class. Execution of elements can vary as and when visitor varies. As per the pattern, element object has to accept the visitor object so that visitor object handles the operation on the element object.

```
import java.util.ArrayList;

public class VisitorPatternOriginal {

  public static void main(String[] args){
    SUTD oneSUTD = new SUTD ();

    ArrayList<Employee> employee = oneSUTD.getEmployee();
    AttributeDisplay disp = new AttributeDisplay();

    // for each element implementing the interface with accept(),
    // pass in the functionality, which reacts to the type of element
    // and executes the appropriate operation
    for (int i =0; i < employee.size(); i++) {
            System.out.printf(employee.get(i).getClass().getSimpleName() + ": ");
            employee.get(i).accept(disp);
    }
  }
}

class SUTD {
  private ArrayList<Employee> employee;

  public SUTD () {
    employee = new ArrayList<Employee>();
    employee.add(new Professor("Sun Jun", 0));
    employee.add(new AdminStaff("Stacey", 5));
    employee.add(new Student("Allan", 3));
  }
  public ArrayList<Employee> getEmployee () {
    return employee;
  }
}

// the visitor interface provides a schematic
// to do different actions depending on
// the type of element
interface Visitor {
  void showAttributes(Professor prof);
  void showAttributes(AdminStaff staff);
  void showAttributes(Student student);
}

// concrete implementation of a visitor
// this visitor displays different attributes
// for different classes passed into it
class AttributeDisplay implements Visitor {

  public void showAttributes(Professor prof) {
        System.out.printf(prof.getName() + ", ");
        System.out.println(prof.getNo_of_publications() + " publications");
  }
  public void showAttributes(AdminStaff staff) {
        System.out.printf(staff.getName() + ", ");
        System.out.println(staff.getEfficiency() + " efficiency");
    }
  public void showAttributes(Student student) {
        System.out.printf(student.getName() + ", ");
        System.out.println(student.getGPA() + " GPA");
    }
```

```
}

// the abstract element to receive the changes
// it has to accept a visitor, which specifies
// the different operations we want to introduce
interface Employee {
  void accept(Visitor v);
}

// concrete implementations of the elements.
// when accept() is called, 'this' is passed to
// the visitor and the appropriate operation is
// executed. Notice that we did not have to modify
// the classes and add each operation as a method.
class Professor implements Employee {
  private String name;
  private int no_of_publications;

  public Professor (String name, int no_of_publications) {
    this.name = name;
    this.no_of_publications = no_of_publications;
  }
  public String getName () {
    return name;
  }
  public int getNo_of_publications() {
    return no_of_publications;
  }
  public void accept(Visitor v) {
    v.showAttributes(this);
  }
}

class AdminStaff implements Employee {
  private String name;
  private float efficiency;

  public AdminStaff (String name, float efficiency) {
    this.name = name;
    this.efficiency = efficiency;
  }
  public String getName() {
    return name;
  }
  public float getEfficiency() {
    return efficiency;
  }
  public void accept(Visitor v) {
    v.showAttributes(this);
  }
}

class Student implements Employee {
  private String name;
  private float GPA;

  public Student (String name, float GPA) {
    this.name = name;
    this.GPA = GPA;
  }
  public String getName() {
    return name;
  }
  public float getGPA() {
    return GPA;
  }
```

```
  public void accept(Visitor v) {
    v.showAttributes(this);
  }
}

// output
// Professor: Sun Jun, 0 publications
// AdminStaff: Stacey, 5.0 efficiency
// Student: Allan, 3.0 GPA
```

2 Testing

2.1 JUnit Overview

JUnit can be used to test an entire object, part of an object, a method or some interacting methods, and interaction between several objects. Test classes include a **test runner** to run the tests (main()), a collection of **test methods**, methods to **set up** the state before/**update** the state after each test and before and after all tests

2.1.1 Assertions

Each test method checks a condition (assertion) and reports to the test runner whether the test failed or succeeded.

- `assertTrue (String message, boolean condition)`
- `assertFalse (String message, boolean condition)`
- `assertEquals (Object expected, Object actual)`
  - Asserts that two objects **are equal**.
- `assertSame (Object expected, Object actual)`
  - Asserts that two objects **refer** to the same object.
- `assertNotSame (Object expected, Object actual)`
  - Asserts that two objects **do not refer** to the same object.
- `assertNull (Object object)`
- `assertNotNull (Object object)`
- `fail (String message)`

2.1.2 Decorators

Different tests can use the objects without sharing state. Objects used in test fixtures should be declared as **instance variables**. Here are some general rules:

- Using `@Test`
  - The Test annotation tells JUnit that the public void method to which it is attached can be run as a test case.
- Objects should be initialized in a `@Before` method
  - Several tests need similar objects created before they can run. Annotating a public void method with `@Before` causes that method to be run before each Test method.
- Objects can be deallocated or reset in an `@After` method
  - If you allocate external resources in a `@Before` method, you need to release them after the test runs. Annotating a public void method with `@After` causes that method to be run after the Test method.
- `@BeforeClass` and `@AfterClass`: Annotating a public static void method with `@BeforeClass` causes it to be run once before any of the test methods in the class. `@AfterClass` will perform the method after all tests have finished. This can be used to perform clean-up activities.

2.1.3 Parameterize your test

Parameterized unit tests call the constructor for each logical set of data values. The same tests are then run on each set of data values.

1. Annotate test class with `@RunWith(Parameterized.class)`.
2. Create a public static method annotated with `@Parameters` that returns a Collection of Objects (as Array) as test data set.
3. Create a public constructor that takes in what is equivalent to one "row" of test data.
4. Create an instance variable for each "column" of test data.
5. Create your test case(s) using the instance variables as the source of the test data.

```
import org.junit.*;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import static org.junit.Assert.*;
import java.util.*;

// include this for running parameterized tests
@RunWith(Parameterized.class)

public class ParameterizedTest {
  public int sum, a, b;

  // classic constructor
  public ParameterizedTest (int sum, int a, int b) {
      this.sum = sum;
      this.a = a;
      this.b = b;
    }

   // decide the list of parameters to be sent to the class
   @Parameters public static Collection<Object[]> parameters() {
        return Arrays.asList (new Object [][] {{0, 0, 0}, {2, 1, 1}});
    }

   // This test is invoked for each of the parameter sent via parameters()
   @Test public void additionTest() {
    assertEquals(sum, Sum.sum(a, b));
   }
}
```

2.2 Mocking objects

Mocking is used to simulate objects that have not been implemented. For example if you have to make a network call, you manually hard-code/return the network response for ALL kinds of network responses and see if your app is working as you expect. You never assume/test a 200 with incorrect data, because that is not your responsibility. Your responsibility is to test your app with a correct 200, or in case of a 400, 500, you test if your app throws the right error.

2.2.1 JMock mocking procedure

1. `Mockery context = new Mockery();`
   - Create a Mockery object. A Mockery represents the context of the object under test: the objects that it communicates with. A Mockery creates mock objects and checks expectations that are set upon those mock objects. By convention the Mockery is stored in an instance variable named context.
2. `Turtle turtle = context.mock(Turtle.class);`
   - NOTE: If a mock object is stored in a local variable, the variable must be declared as final so that it can be referred to from within expectation blocks.
3. ```
   context.checking(new Expectations() {{
     ... expectations go here ...
   }});
   ... call code being tested ...
   context.assertIsSatisfied();
   ```
   - An expectations block can contain any number of expectations. A test can contain multiple expectation blocks. Expectations in later blocks are appended to those in earlier blocks. Expectations can be interspersed with calls to the code under test.
   - Invocation count:
     - `oneOf` The invocation is expected once and once only.
     - `exactly(n).of` The invocation is expected exactly n times. Note: one is a convenient shorthand for exactly(1).
     - `atLeast(n).of` The invocation is expected at least n.
     - `atMost(n).of` The invocation is expected at most n times.
     - `between(min, max).of` The invocation is expected at least min times and at most max times.
     - `allowing` The invocation is allowed any number of times but does not have to happen.
     - `ignoring` The same as allowing. Allowing or ignoring should be chosen to make the test code clearly express intent.
     - `never` The invocation is not expected at all. This is used to make tests more explicit and so easier to understand.
   - Actions:
     - `will(returnValue(v))` Return v to the caller.
     - `will(returnIterator(c))` Return a new iterator over collection c on each invocation.
     - `will(returnIterator(v1, v2, ..., vn))` Return a new iterator over elements v1 to vn on each invocation.
     - `will(throwException(e))` Throw e to the caller.
     - `will(doAll(a1, a2, ..., an))` Do all actions a1 to an on every invocation.
     - For example, `oneOf (calculator).add(1, 1); will(returnValue(2));`
   - Sequences, for invocations in an order:
   ```

```
        — final Sequence sequence-name = context.sequence("sequence-name");
          oneOf (turtle).forward(10); inSequence(drawing);
          oneOf (turtle).turn(45); inSequence(drawing);
```

```
import org.junit.Test;
import org.jmock.Mockery;
import org.jmock.integration.junit4.JUnit4Mockery;

import org.jmock.Expectations;

public class TestWithMock {
    @Test
    public void testCalculatingMachine() {
        // create the context to mock classes
        Mockery context = new JUnit4Mockery();

        // mock the classes that are not implemented yet
        final Printer printer = context.mock(Printer.class);
        final Calculator calculator = context.mock(Calculator.class);

        //provide expectations in the test execution
        //@oneOf: one invocation of the function is expected
        //@never: the invocation must never happen
        //@will(returnValue): will expect the return value specified by the argument
        context.checking(new Expectations() {{
            oneOf(calculator).add(1, 2);
            // never(printer).print("result is 3");
            // will(returnValue(3));
            // oneOf(printer).print("result is 3");
        }});

        // what follows is the test execution
        CalculatingMachine machine = new CalculatingMachine(printer, calculator);
        machine.processAdd(1, 2);

        // fails the above test execution if any expectation is violated
        context.assertIsSatisfied();
    }
}
```

2.3 Fuzzing

2.3.1 Regular fuzzing

```
#include <stdlib.h>
#include <assert.h>
#include <time.h>
#include <stdio.h>

int main() {
  char* input;
  int index;

  // strings of 32
  srand(time(NULL));
  int string_length = 32;

  //allocate memory for the input
  input = (char *) malloc ((string_length+1) * sizeof(char));

  //generate a random character at each location of the string
  for (index = 0; index < string_length; index++) {
    double between_0_and_1 = (double)rand() / (double)RAND_MAX;
    //generate a character between ASCII 32 and 128
    input[index] = (int)(between_0_and_1 * 96 + 32);
  }
  input[string_length] = '\0';
  // here is the input string to fuzz
  printf("%s\n", input);
}
```

2.3.2 Mutation fuzzing

Mutation fuzzing includes flipping a bit, trimming, swapping characters, and inserting characters.

```
from random import randint

inp = open("generalized_fuzzer_input")
line_array = inp.read().splitlines()

def swap(line):
    if (len(line) < 2):
        print("can't swap if only 1 char")
        return
    # implementation from ex.3 (mutation.py)
    position = randint(0,len(line)-2)
    left = line[:position] + line[position+1]
    right = line[position] + line[position+2:]
    print("swapped: "+line+ " -> "+left+right)

def trim(line):
    if (len(line) < 2):
        print("can't trim if only 1 char")
        return
    position = randint(0,len(line)-2)
    trimmed = "trimmed: " + line + " -> " + line[:position]
    print(trimmed)

def flip(line):
    flipped = ""
    for char in line:
        # choose random index to flip
        index = randint(0,6)
        binary = bin(ord(char))[2:]
        # ascii space is 32=2^5, so need to add a 0 in front
        if (len(binary) == 6):
            binary = "0" + binary
        if (binary[index] == '0'):
            binary = binary[:index] + '1' + binary[index+1:]
        elif (binary[index] == '1'):
            binary = binary[:index] + '0' + binary[index+1:]
        flipped += chr(int(binary,2))
    print("flipped: " + line + " -> "+ flipped)

# randomly choose a mutation
mutations = {
    0: swap,
    1: trim,
    2: flip
}

for line in line_array:
    mut = randint(0,len(mutations)-1)
    mutations[mut](line)
```

2.4 Web Testing

Selenium API allows access to the DOM and its content. Selected classes and methods:

1. **public interface** WebElement
   Represents an HTML element. Generally, all interesting operations to do with interacting with a page will be performed through this interface. All method calls will do a freshness check to ensure that the element reference is still valid. This essentially determines whether or not the element is still attached to the DOM. If this test fails, then an StaleElementReferenceException is thrown, and all future calls to this instance will fail.

   - **void** clear() If this element is a text entry element, this will clear the value.
   - **void** click() Click this element.
   - WebElement findElement(By by) Find the first WebElement using the given method.
   - List<WebElement> findElements(By by) Find all elements within the current context using the given mechanism.
   - String getAttribute(String name) Get the value of the given attribute of the element.
   - String getCssValue(String propertyName) Get the value of a given CSS property.
   - Point getLocation() Where on the page is the top left-hand corner of the rendered element?
   - getRect() Returns the location and size of the rendered element.
   - Dimension getSize() What is the width and height of the rendered element?
   - String getTagName() Get the tag name of this element.
   - String getText() Get the visible text.
   - isDisplayed() Is this element displayed or not? This method avoids the problem of having to parse an element's "style" attribute.
   - **boolean** isEnabled() Is the element currently enabled or not? This will generally return true for everything but disabled input elements.
   - **boolean** isSelected() Determine whether or not this element is selected or not.
   - **void** sendKeys(CharSequence... keysToSend) Use this method to simulate typing into an element, which may set its value.
   - **void** submit() If this current element is a form, or an element within a form, then this will be submitted to the remote server.

2. **public interface** WebDriver
   The main interface to use for testing, which represents an idealised web browser. Key methods are get(String), which is used to load a new web page, and the various methods similar to findElement(By), which is used to find WebElements.

   - **void** close() Close the current window, quitting the browser if it's the last window currently open.
   - WebElement findElement(By by) Find the first WebElement using the given method.
   - List<WebElement> findElements(By by) Find all elements within the current page using the given mechanism.
   - **void** get(String url) Load a new web page in the current browser window.
   - String getCurrentUrl() Get a string representing the current URL that the browser is looking at.
   - String getPageSource() Get the source of the last loaded page.
   - String getTitle() The title of the current page.

- `String getWindowHandle()` Return an opaque handle to this window that uniquely identifies it within this driver instance.
- `Set<String> getWindowHandles()` Return a set of window handles which can be used to iterate over all open windows of this WebDriver instance by passing them to switchTo().WebDriver.Options.window()
- `WebDriver.Options manage()` Gets the Option interface
- `WebDriver.navigate()` An abstraction allowing the driver to access the browser's history and to navigate to a given URL.
- **`void`** `quit()` Quits this driver, closing every associated window.
- `WebDriver.TargetLocator switchTo()` Send future commands to a different frame or window.

3. **`public interface`** `ExpectedCondition<T>`
   Models a condition that might reasonably be expected to eventually evaluate to something that is neither null nor false. Examples would include determining if a web page has loaded or that an element is visible. Consider the Google login example. We do not need to wait always 10 seconds. We only need to wait until the password field becomes visible in the page.

   - ```
     try {
       WebDriverWait wait = new WebDriverWait(driver, 10);
       // wait only until the password element becomes visible
       wait.until(ExpectedConditions.elementToBeClickable(By.name("password")));
       // now locate the password field in the current page
       WebElement password = driver.findElement(By.name("password"));
       // send password
       password.sendKeys(myPassword);
       // login and :)
       nextButton = driver.findElement(By.id("passwordNext"));
       nextButton.click();
     } catch (Exception NoSuchElementException) {
       System.out.println("login name invalid");
     }
     ```

Random link clicking is implemented through WebDriver and clicking a random WebElement:

```
import org.openqa.selenium.By;
import org.openqa.selenium.StaleElementReferenceException;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

import java.util.Random;

public class ISTDWebsiteClickRandomLink {

  public static void main(String[] args) throws InterruptedException {
    System.setProperty("webdriver.chrome.driver","/Users/Joel/Desktop/SUTD/Term 5/50.003
    Elements of Software Construction/chromedriver.exe");
    WebDriver driver = new ChromeDriver();

    driver.get("https://istd.sutd.edu.sg");
        Random rand = new Random();

    while (true) {
          // get all the links
      java.util.List<WebElement> links = driver.findElements(By.tagName("a"));
      System.out.println(links.size());

      // click a random link
          int nextLink = rand.nextInt(links.size());
```
```

```
      System.out.println("*** Navigating to" + " " + links.get(nextLink).getAttribute("href
    "));
            if (links.get(nextLink).getAttribute("href") == null) continue;

            try {
                driver.navigate().to(links.get(nextLink).getAttribute("href"));
                Thread.sleep(3000);
                //driver.navigate().back();
                System.out.println("*** Navigated to" + " " + links.get(nextLink).
    getAttribute("href"));
                //staleElementLoaded = false;
            } catch (StaleElementReferenceException e) {
                //staleElementLoaded = true;
            }
        }
    }
}
```

3 Code Refactoring

The more lines found in a method, the harder it is to figure out what the method does. This is the main reason for this refactoring. Besides eliminating rough edges in your code, extracting methods is also a step in many other refactoring approaches. Refactoring gives you more readable code, less code duplication. Often the code that is found in a method can be reused in other places in your program. Also isolates independent parts of code, meaning that errors are less likely (such as if the wrong variable is modified).

3.1 Code smells

3.1.1 Long method

- As a rule of thumb, if you feel the need to comment on something inside a method, you should take this code and put it in a new method. Even a single line can and should be split off into a separate method, if it requires explanations. And if the method has a descriptive name, nobody will need to look at the code to see what it does.
- Solution: Extract method. Change from

```
void printOwing() {
  printBanner();

  //print details
  System.out.println("name: " + name);
  System.out.println("amount: " + getOutstanding());
}
```

- to:

```
void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}

void printDetails(double outstanding) {
  System.out.println("name: " + name);
  System.out.println("amount: " + outstanding);
}
```

3.1.2 Repeated code

- There is also more subtle duplication, when specific parts of code look different but actually perform the same job. This kind of duplication can be hard to find and fix. Sometimes duplication is purposeful. When rushing to meet deadlines and the existing code is "almost right" for the job, novice programmers may not be able to resist the temptation of copying and pasting the relevant code. And in some cases, the programmer is simply too lazy to de-clutter.
- Use the same approach as long method to solve it.

3.1.3 Black Hole class/Large class

- As is the case with long methods as well, programmers usually find it mentally less taxing to place a new feature in an existing class than to create a new class for the feature. When a class is wearing too many (functional) hats, think about splitting it up.

- Single-responsibility classes are more reliable and tolerant of changes. For example, say that you have a class responsible for ten different things. When you change this class to make it better for one thing, you risk breaking it for the nine others.
- To solve this, create new classes. Create a relationship between the old class and the new one. Optimally, this relationship is unidirectional; this allows reusing the second class without any issues. Nonetheless, if you think that a two-way relationship is necessary, this can always be set up.

3.1.4 Data class (Not to be confused with *Data clump*)

- A data class refers to a class that contains only fields and crude methods for accessing them (getters and setters). These are simply containers for data used by other classes. These classes do not contain any additional functionality and cannot independently operate on the data that they own.
- Fixing this improves understanding and organization of code. Operations on particular data are now gathered in a single place, instead of haphazardly throughout the code.
- After the class has been filled with well thought-out methods, you may want to get rid of old methods for data access that give overly broad access to the class data.
- Example:

```
class Point3D {
  int x;
  int y;
  int z;

  public Point3D(int x, int y, int z) {
    this.x = x;
    this.y = y;
    this.z = z;
  }
  public void setX(int x) {this.x = x;}
  public void setY(int y) {this.y = y;}
  public void setZ(int z) {this.z = z;}
  public int getX(int x) {return x;}
  public int getY(int y) {return y;}
  public int getZ(int z) {return z;}
}
```

3.1.5 Data clump

- Sometimes different parts of the code contain similar groups of variables (such as parameters for connecting to a database). These clumps (e.g. arg1, arg2, arg3, ..., which are always found together) should be turned into a class of its own.
- If you want to make sure whether or not some data is a data clump, just delete one of the data values and see whether the other values still make sense. If this is not the case, this is a good sign that this group of variables should be combined into an object.
- Refactor from:

```
void demoClump() {
  int param1 = 0;
  String param2 = "yes";
```

```
  boolean param3 = true;
  action (int p1, String p2, boolean p3);
}
```

- to:

```
class NotClumped {
  int param1 = 0;
  String param2 = "yes";
  boolean param3 = true;
  action () { // use vars here }
}

... then the class can be used as ...
NotClumped nc = new NotClumped(0, "yes", true);
nc.action();
```

3.1.6 Long parameter

- A long list of parameters might happen after several types of algorithms are merged in a single method. A long list may have been created to control which algorithm will be run and how.
- Solution is to only pass parameters that are needed. Let's say we are passing `basePrice`, `discountAmount`, and `fees` to a method called `discountedPrice()` which calculates the new discounted price. `discountAmount` and `fees` actually do not need to be passed in and can be obtained in the method itself.

3.1.7 Shotgun surgery

- Making any modifications requires that you make many small changes to many different classes. This happens when a single responsibility has been split up among a large number of classes.
- Move methods and fields to force existing class behaviors into a single class. If there is no class appropriate for this, create a new one.

3.1.8 Feature envy

- A method accesses the data of another object more than its own data. This smell may occur after fields are moved to a data class. If this is the case, you may want to move the operations on data to this class as well.
- Solve this by moving the methods from the envied class to the envious class.
- Example:

```
class Project {
}

class GirlFriend {
  Project myProject;
  public GirlFriend(Project myProject) {this.myProject = myProject;}
  public Project getProject() {return this.myProject;}
}

public class LostInLove {
  GirlFriend myGirlfriend;
```

```
  Project myProject;

  public LostInLove(GirlFriend myGirlfriend, Project myProject) {
    this.myGirlfriend = myGirlfriend;
    this.myProject = myProject;
  }

  public void doHerProject() {
    Project herProject = myGirlfriend.getProject();
    workOnProject(herProject);
  }

  public void doProject() {workOnProject(myProject);}

  public void workOnProject(Project p) {}
}
```

3.1.9 Message chains

- Message chains are something like `a.getB().getC().getValue()`. A message chain occurs when a client requests another object, that object requests yet another one, and so on. These chains mean that the client is dependent on navigation along the class structure. Any changes in these relationships require modifying the client.
- A middle man is a method that shortens the message chain: `a.getB().getCValue()`. One benefit of using middle men instead of message chains is that you have to provide fewer mocks when doing unit testing. Classes become really hard to test when you have to provide mocks not only for their direct dependencies but also their indirect ones.

3.1.10 Speculative generality

- There is an unused class, method, field or parameter. Sometimes code is created just in case to support anticipated future features that never get implemented. As a result, code becomes hard to understand and support.
- We should avoid over engineering based on unlikely generalization, it violates the agile development principle. We should concentrate on the features needed, throw away all other features: the art of maximizing work not done.

3.1.11 Refused bequest

- If a subclass uses only some of the methods and properties inherited from its parents, the hierarchy is off-kilter. The unneeded methods may simply go unused or be redefined and give off exceptions. Someone was motivated to create inheritance between classes only by the desire to reuse the code in a superclass. But the superclass and subclass are completely different.
- If inheritance makes no sense and the subclass really does have nothing in common with the superclass, eliminate inheritance. Create a field and put a superclass object in it, delegate methods to the superclass object (Another way is to shift the common class into another abstract class, then inherit for one and extend for another).

4 Requirement and Analysis

1. Pre-conditions are the things that must be true before a method is called. The method tells clients "this is what I expect from you".
2. Post-conditions are the things that must be true after the method is complete. The method tells clients "this is what I promise to do for you".
3. Invariants are the things that are always true and won't change. The method tells clients "if this was true before you called me, I promise it'll still be true when I'm done".
4. Example: "Given a **semi-prime** (pre-condition), your program outputs its **prime factors** (post-condition) **within 6 days** (non-functional requirement)."

5 Implementing Concurrency

5.1 Threading

In this course, concurrency can be achieved in a few ways:

1. Implementing the `Runnable` interface.
2. Extending the `Thread` class.
3. Combine `Runnable` and `Thread`.
4. `Callables` and `FutureTasks` (From the javadoc: "A cancellable asynchronous computation"). The integration of timeouts, proper cancelling and the thread pooling of the modern concurrency support are all much more useful than piles of raw `Threads`.

In most cases choose `Runnable` rather than `Thread` because it allows you to keep your work only loosely coupled with your choice of concurrency.

5.1.1 Using the Thread class

Workflow using Thread[] First set up a `Thread[] arr.` `for i = 0 to i < arr.length`, do `arr[i].start()`. Then iterate through them and call `arr[i].join()` as necessary. Here you have to handle `InterruptException`.

Creating the Thread There are two ways to create a new thread of execution. One is to declare a class to be a **subclass of Thread**. This subclass should override the run method of class Thread. The other way to create a thread is to declare a class that **implements the Runnable interface**. That class then implements the run method. An instance of the class can then be allocated, passed as an argument when creating Thread, and started. Important methods in the Thread class:

- `void interrupt()` Interrupts this thread.
- `static boolean interrupted()` Tests whether the current thread has been interrupted.
- `boolean isInterrupted()` Tests whether this thread has been interrupted.
- `join(long millis)` [Optional: milis] Waits at most millis milliseconds for this thread to die.
- `run()` If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called; otherwise, this method does nothing and returns.
- `start()` Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.

- `yield()` A hint to the scheduler that the current thread is willing to yield its current use of a processor.

Notes:

1. interrupted() is **static** and checks the current thread. isInterrupted() is an **instance method** which checks the Thread object that it is called on. A common error is to call a static method on an instance.
2. When we call the start() method, it **starts the thread** which executes the run method. If you call run() directly you don't start the thread, you just execute the method in the current running thread.
3. Using yield() to give up CPU to another thread:
```
while (!ready) {
  Thread.yield();
}
```

5.1.2 Combining Runnable interface and Thread class

```
class MyRunnable implements Runnable {
  private CachedFactorizerSafe factorizer;

  public MyRunnable (CachedFactorizerSafe factorizer) {
    this.factorizer = factorizer;
  }

  public void run () {
    Random random = new Random ();
    factorizer.service(random.nextInt(100));
  }
}
...
CachedFactorizerSafe factorizer = new CachedFactorizerSafe();
Thread tr1 = new Thread (new MyRunnable(factorizer));
tr1.start();
```

5.1.3 Killing threads

A thread ends when it reaches the end of `run()`. For manual killing, always use `Thread.interrupt()`. Find a way to communicate between threads in order to stop them properly. Some ways:

- Use a static boolean variable (Consider using volatile, so that it is visible across threads).
- wait(), notify(), and notifyAll(). If thread holds a lock, then wait() puts it to the wait set. On notify(), a single thread is woken from the set. On notifyAll(), all are woken and they will end up reevaluating their conditions.

Why is Thread.stop deprecated? Because it is inherently unsafe. Stopping a thread causes it to unlock all the monitors that it has locked. (The monitors are unlocked as the ThreadDeath exception propagates up the stack.) If any of the objects previously protected by these monitors were in an inconsistent state, other threads may now view these objects in an inconsistent state. Such objects are said to be damaged. When threads operate on damaged objects, arbitrary behavior can result. This behavior may be subtle and difficult to detect, or it may be pronounced. Unlike other unchecked exceptions, ThreadDeath kills threads silently; thus, the user has no warning that his program may be corrupted. The corruption can manifest itself at any time after the actual damage occurs, even hours or days in the future.

Why are Thread.suspend and Thread.resume deprecated? Thread.suspend is inherently deadlock-prone. If the target thread holds a lock on the monitor protecting a critical system resource when it is suspended, no thread can access this resource until the target thread is resumed. If the thread that would resume the target thread attempts to lock this monitor prior to calling resume, deadlock results. Such deadlocks typically manifest themselves as "frozen" processes.

What about Thread.destroy? Thread.destroy was never implemented and has been deprecated. If it were implemented, it would be deadlock-prone in the manner of Thread.suspend. (In fact, it is roughly equivalent to Thread.suspend without the possibility of a subsequent Thread.resume.)

6 Concurrency Problems and Solutions

6.1 Deadlock

6.1.1 How to identify deadlock and other liveness problems

At runtime Deadlock only occurs when a thread fails to lock a mutex. Suspend all threads and check for cyclic mutex dependency by performing a tree traversal of the dependencies. An expensive way to solve this problem is to checkpoint address space to disk everytime a mutex is grabbed. As soon as there is deadlock, it can rollback and then introduce some rule in the scheduler to fix the order. This can't work for irreversible actions like sending packets over the network. Another solution is to kill the threads involved in the deadlock, after ensuring that the data invariants are enforced.

In the code Can suffer from deadlock, livelock and/or starvation. Livelock is the situation where a thread, while not blocked, still cannot make progress because it keeps retrying an operation that will always fail. Starvation occurs when a thread is denied access to resources it needs in order to make progress. It's often caused by use of thread priority or executing infinite loops with a lock held. Common culprits in code:

- **Nested Locks:** This is the most common reason for deadlocks, avoid locking another resource if you already hold one. It's almost impossible to get deadlock situation if you are working with only one object lock.
- **Locking more than required:** You should acquire lock only on the resources you have to work on, if we are only interested in one field, then we should lock only that specific field not the complete object.
- **Waiting indefinitely:** You can get deadlock if two threads are waiting for each other to finish indefinitely using thread join. If your thread has to wait for another thread to finish, it's always best to use join with maximum time you want to wait for thread to finish.
- **Thread priority:** Avoid using thread priority, since they increase platform dependence and can cause liveness problems.
- **Lock releasing:** You can also easily enter a livelock condition from here. If each philosopher puts down the chopstick on his right and tries to pick up the one on the left, then they're stuck that way. They may put the left one down and pick up the one on the right again, oscillating between two states without actually managing to eat.

Solutions to the Dining Philosophers' Problem, but applicable elsewhere:

- Don't allow all philosophers to sit and eat/think at once.
- Pick up both chopsticks in a critical section
- Alternate choice of first chopstick

- Enforce a locking order, in which you can acquire two locks only by doing so in a specified order. If you acquire one lock and fail to acquire the other, then you must release the first lock and try again (if each chopstick had a number, and they all had to be picked up in order. In this case, all of the philosophers would try to pick up the one on their right, except for the one sitting between the chopsticks with the highest number on one side and the lowest number on the other side. He would have to wait until the person using the chopstick on his left had finished before he could pick up the one on his right. Or he would get there first. In both cases, at least one person would be able to pick up both chopsticks. Once he'd finished eating, at least one more person would be able to start).
- Lock one and trylock the next (More under Reentrant locking below).

6.2 Thread Safety Toolbox

If an object is to be shared by multiple threads, it must be thread safe. A class is thread-safe if no set of operations performs sequentially or concurrently on instances of a thread-safe class can cause an instance to be in an invalid state (defined by the specifications). What are some indicators of thread-safe code? No race conditions, no visibility issues, no execution ordering problems, and no deadlocks. Optionally, they should be efficient. But correctness is more important.

6.2.1 General rules for thread safety

- Update related state variables in a single atomic operation.
- For each mutable variable that may be accessed by more than one thread, all assesses to that variable must be performed with the same lock held.
- Every shared, mutable variable should be guarded by exactly one lock. Make it clear to maintainers which lock that is.
- For every invariant that involves more than one variable, all the variables involved in that invariant must be guarded by the same lock.

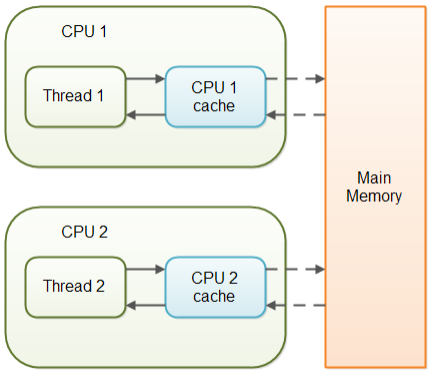
6.2.2 Volatile variables

Use the `volatile` keyword to avoid threads working on stale cached copies of variables. An update to a volatile variable is propagated predictably to other threads. All reads and writes will go straight to main memory. The main differences between `volatile` and `synchronized` are:

- A primitive variable may be declared volatile (whereas you can't synchronize on a primitive with synchronized);
- An access to a volatile variable never has the potential to block: we're only ever doing a simple read or write, so unlike a synchronized block we will never hold on to any lock;
- Because accessing a volatile variable never holds a lock, it is not suitable for cases where we want to read-update-write as an atomic operation (unless we're prepared to miss an update);
- A volatile variable that is an object reference may be null (because you're effectively synchronizing on the reference, not the actual object).

A volatile variable is **no longer enough** to guarantee correct visibility as soon as a thread needs to first read the value of a volatile variable, and based on that value generate a new value for the shared volatile variable. The short time gap in between the reading of the volatile variable and the writing of its new value, creates an race condition where multiple threads might overwrite each other's values.

Imagine if Thread 1 **reads a shared counter variable with the value 0** into its CPU cache, increment it to 1 and **not write the changed value back into main memory**. Thread 2 could then read the same counter variable from main memory where the **value of the variable is still 0**, into its own CPU cache. Thread 2 could then also increment the counter to 1, and also not write it back to main memory. This situation is illustrated in the diagram below:



6.2.3 Atomic package

Use this when we are modifying values of variables concurrently. For example, `AtomicInteger`, an object representing an int value that may be updated atomically:

- `int accumulateAndGet(int x, IntBinaryOperator accumulatorFunction)` Atomically updates the current value with the results of applying the given function to the current and given values, returning the updated value.
- `int addAndGet(int delta)` Atomically adds the given value to the current value.
- `boolean compareAndSet(int expect, int update)` Atomically sets the value to the given updated value if the current value == the expected value.
- `int decrementAndGet()` Atomically decrements by one the current value.
- `double doubleValue()` Returns the value of this AtomicInteger as a double after a widening primitive conversion.
- `float floatValue()` Returns the value of this AtomicInteger as a float after a widening primitive conversion.
- `int get()` Gets the current value.
- `int getAndAccumulate(int x, IntBinaryOperator accumulatorFunction)` Atomically updates the current value with the results of applying the given function to the current and given values, returning the previous value.
- `int getAndAdd(int delta)` Atomically adds the given value to the current value.
- `int getAndDecrement()` Atomically decrements by one the current value.
- `int getAndIncrement()` Atomically increments by one the current value.
- `int getAndSet(int newValue)` Atomically sets to the given value and returns the old value.
- `int getAndUpdate(IntUnaryOperator updateFunction)` Atomically updates the current value with the results of applying the given function, returning the previous value.

- `int incrementAndGet()` Atomically increments by one the current value.
- `int intValue()` Returns the value of this AtomicInteger as an int.
- `void lazySet(int newValue)` Eventually sets to the given value.
- `long longValue()` Returns the value of this AtomicInteger as a long after a widening primitive conversion.
- `void set(int newValue)` Sets to the given value.
- `String toString()` Returns the String representation of the current value.
- `int updateAndGet(IntUnaryOperator updateFunction)` Atomically updates the current value with the results of applying the given function, returning the updated value.
- `boolean weakCompareAndSet(int expect, int update)` Atomically sets the value to the given updated value if the current value == the expected value.

Usage in threads:

```
class Incrementer extends Thread {
  public void run () {
      myAtomicInteger.incrementAndGet();
  }
}
```

6.2.4 Intrinsic locking: Synchronized methods

Synchronized methods At its simplest level, a block of code that is marked as synchronized in Java tells the JVM: "only let one thread in here at a time". Marking a method as `synchronized` makes the compiler append instructions to acquire the lock on the specified object before executing the code, and release it afterwards (either because the code finishes normally or abnormally). Intrinsic locks are reentrant.

Synchronized blocks Every Java `Object` can implicitly act as a lock for purposes of synchronization, using the `synchronized` block:

```
synchronized (lock) {
  //Access shared state guarded by lock
}
```

NOTE: Always be careful of the **locking policy** and be aware of what you are locking. If you lock on two different objects when what you really wanted was to synchronize on the same object, then the locking is useless.

6.2.5 Private locking: ReentrantLock

```
class Withdrawer extends Thread {
  public void run () {
    reentrantMutex.lock();
    if (saving >= 1000) {
      saving = saving - 1000;
      cash = cash + 1000;
    }
    reentrantMutex.unlock();
  }
}
```

Usage and Conditions When a thread requests a lock that is already held by another thread, the requesting thread blocks. Also, possible to wait on certain conditions using `Condition c = reentrantMutex.newCondition()` and then using `c.await()`, `c.awaitInterruptibly()` to cause the current thread to wait until it is signalled or interrupted, and `c.signal()` and `c.signalAll()` to wake up one/all waiting thread(s) for that particular condition.

tryLock() Acquires the lock if it is not held by another thread and returns immediately with the value true, setting the lock hold count to one. Even when this lock has been set to use a fair ordering policy, a call to `tryLock()` will immediately acquire the lock if it is available, whether or not other threads are currently waiting for the lock. This "barging" behavior can be useful in certain circumstances, even though it breaks fairness. If you want to honor the fairness setting for this lock, then use `tryLock(0, TimeUnit.SECONDS)` (acquires the lock if it's available within the given waiting time), which is almost equivalent (it also detects interruption). If the current thread already holds this lock then the hold count is incremented by one and the method returns true. If the lock is held by another thread then this method will return immediately with the value false. Here is an example that uses `ReentrantLock` and `tryLock()`, to solve the Dining Philosophers' Problem:

```
import java.util.Random;
import java.util.concurrent.locks.ReentrantLock;


// Locking policy: Each Fork1 object has a reentrant lock that
// abstracts the availability of the lock. Since deadlock can only happen
// if each Philosopher is holding on to one fork and waiting for a held Fork1,
// if all 5 Philosophers hold on to forks on the same hand, e.g. left,
// no forks would be left. If they try to get a Fork on the right,
// then they would fail and release the left one. This solution prevents
// deadlock and can be implemented either on the left or right hands.

public class DiningPhil1 {

  private static int N = 5;

  public static void main (String[] args) throws Exception {
    Philosopher1[] phils = new Philosopher1[N];
    Fork1[] forks = new Fork1[N];

    for (int i = 0; i < N; i++) {
      forks[i] = new Fork1(i);
    }

    for (int i = 0; i < N; i++) {
      phils[i] = new Philosopher1 (i, forks[i], forks[(i+N-1)%N]);
      phils[i].start();
    }
  }
}

class Philosopher1 extends Thread {
  private final int index;
  private final Fork1 left;
  private final Fork1 right;

  public Philosopher1 (int index, Fork1 left, Fork1 right) {
    this.index = index;
    this.left = left;
    this.right = right;
  }

  public void run() {
    Random randomGenerator = new Random();
    try {
      while (true) {
```

```
        // Lock left, then tryLock right.
        Thread.sleep(randomGenerator.nextInt(100)); //not sleeping but thinking
        System.out.println("Phil " + index + " finishes thinking.");
        left.lock.lock();
        System.out.println("Phil " + index + " locks left.");
        left.pickup();
        System.out.println("Phil " + index + " picks up left Fork1.");
        System.out.println("Phil " + index + " tries to lock right.");
        boolean lockRight;
        if (lockRight = right.lock.tryLock()) {
          System.out.println("Phil " + index + " locks right.");
          right.pickup();
          System.out.println("Phil " + index + " picks up right Fork1.");
        }
        Thread.sleep(randomGenerator.nextInt(100)); //eating
        System.out.println("Phil " + index + " finishes eating.");
        left.putdown();
        System.out.println("Phil " + index + " puts down left Fork1.");
        left.lock.unlock();
        System.out.println("Phil " + index + " unlocks left.");
        if (lockRight) {
          right.putdown();
          System.out.println("Phil " + index + " puts down right Fork1.");
          right.lock.unlock();
          System.out.println("Phil " + index + " unlocks right.");
        }
      }
    } catch (InterruptedException e) {
      System.out.println("Don't disturb me while I am sleeping, well, thinking.");
    }
  }
}

class Fork1 {
  private final int index;
  private boolean isAvailable = true;
  public ReentrantLock lock = new ReentrantLock();

  public Fork1 (int index) {
    this.index = index;
  }

  public synchronized void pickup () throws InterruptedException {
    while (!isAvailable) {
      wait();
    }

    isAvailable = false;
    notifyAll();
  }

  public synchronized void putdown() throws InterruptedException {
    while (isAvailable) {
      wait();
    }

    isAvailable = true;
    notifyAll();
  }

  public String toString () {
    if (isAvailable) {
      return "Fork " + index + " is available.";
    }
    else {
      return "Fork " + index + " is NOT available.";
```

```
    }
  }
}
```

6.3 Locking Policy

6.3.1 How do I make something Thread-safe?

1. Ask, what is the **Object's state** and what **constraints** does it have?
   (a) All mutable variables, if they are all primitive.
   (b) All mutable variables + references to other Objects, otherwise.
   (c) Reduce this state space by using final variables.
   (d) Identify if things like atomicity and encapsulation are required.
2. How do I establish and implement the policy for concurrent access to these variables?
   (a) Guard each mutable variable with one lock.
   (b) Guard related variables with the same locks.
   (c) Make sure every access of any variable is guarded by the lock according to the policy.
   (d) Make sure access of the related variables in the same method is in synchronized block.
   (e) Add waiting (and notify) to handle pre-conditions.
3. Can you **extend another class** that's already known to be Thread-safe?
   (a) If there exists such a class supporting most to all of the needed operations.
   (b) If you know the locking policy of that class.
   (c) **More fragile than modifying the class directly**, because the implementation of the synchronization policy is now distributed over multiple, separately maintained source files.
4. If you don't have access to such a class, can you use **Client-side locking**?
   (a) If we know the client's locking policy. Client-side locking is when we outsource the locking and unlocking to the program that is requesting the resource.
   (b) The following example is **wrong**:

```
public class ListHelper<E> {
  public java.util.List<E> list = Collections.synchronizedList(new ArrayList<E>());

  public synchronized boolean putIfAbsent(E x) {
    boolean absent = !list.contains(x);
    if (absent) {
        list.add(x);
    }
    return absent;
  }
}
```

   (c) Here, the method `putIfAbsent()` is `synchronized`, but on the superclass instead of on the list. Multiple threads can still modify the list through other methods without acquiring the lock on the list object. The solution is to wrap the contents of the method with the block `synchronized (list)` instead.
   (d) In general, **Client-side locking is even more fragile** than extending the class because it distributes the locking policy for a class into classes that are totally unrelated.

5. **Composition:** Or I can make a **data structure wrapper** around `List<E>`, for example, where **everything** is `synchronized`. That way I don't need to care if `List<E>` is Thread-safe or not, I can just interface with it using my `synchronized` methods.

   (a) Yes, but then the whole data structure takes a performance hit. Imagine if every method accessing, reading, writing, modifying a data structure is made `synchronized`, and you have say, thousands of threads trying to do operations on that structure. It's going to slow down quite a bit.

6. Are the instances confined?

   (a) Objects mustn't escape their scope. Escaping includes public variables lying around, methods locking on wrong objects, etc.

   (b) Things such as `public int topOfStack` shouldn't be lying around in public. Some other program can modify it directly even without using the thread-safe method that you went through great pains to implement.

   (c) How to confine? Use `private`, make variables local, confine variables to the specific threads if they are not supposed to be shared.

6.4 Java's Concurrency Library

Synchronized Collections, Concurrent Collections, and Synchronizers.

6.4.1 Synchronized collections

Why are `Vector` and `Hashtable` deprecated? All of the get(), set() methods are synchronized, so you can't have fine grained control over synchronization. Vector is essentially an early ArrayList that has all its methods synchronized, and Hashtable is an early Hashmap that's synchronized whereas the new HashMap is not. They are not efficient if the data structure starts getting large and because of using `synchronized` everywhere they slow down quite a bit.

Internal fail-fast using Iterators When you do `for (Object obj : objList)` where `objList` is a `List<Object>`, internally, javac generates code that uses an Iterator, repeatedly calling hasNext and next to iterate the list. If the Collection will be changed while some thread is traversing over it using Iterator, the iterator.next() will throw ConcurrentModificationException (an int variable modCount is defined: modCount provides the number of times list size has been changed. modCount value is used in every next() call to check for any modifications in a function checkForComodification()). Therefore you have to lock either when the specific method to modify is called, or before the entire process of iteration, depending on the specification. But this is bad because locking the entire data structure removes the benefits of multithreading. One alternative is to clone the collection, lock and iterate the copy.

6.4.2 Java 5.0: Concurrent collections

ConcurrentHashMap ConcurrentHashMap uses **lock striping**. An array of 16 locks, with each lock guarding $\frac{1}{16}$ of the buckets (bucket N is guarded by lock (N%16)). The iterators returned by ConcurrentHashMap are weakly consistent (i.e., it is OK to modify the collection while iterating through it) instead of fail-fast, and it cannot be locked for exclusive access. Useful methods in ConcurrentHashMap:

- `void clear()` Removes all of the mappings from this map.
- `V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)` If the specified key is not already associated with a value, attempts to compute its value using the given mapping function and enters it into this map unless null.

- `V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)` If the value for the specified key is present, attempts to compute a new mapping given the key and its current mapped value.
- `V get(Object key)` Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
- `V put(K key, V value)` Maps the specified key to the specified value in this table.
- `V putIfAbsent(K key, V value)` If the specified key is not already associated with a value, associate it with the given value.
- `V remove(Object key)` Removes the key (and its corresponding value) from this map.
- **`boolean`** `remove(Object key, Object value)` Removes the entry for a key only if currently mapped to a given value.
- `V replace(K key, V value)` Replaces the entry for a key only if currently mapped to some value.
- **`boolean`** `replace(K key, V oldValue, V newValue)` Replaces the entry for a key only if currently mapped to a given value.

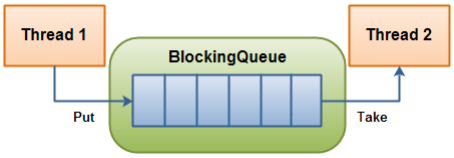
Using ConcurrentHashMap's `putIfAbsent()` with FutureTask:

```
// a ConcurrentHashMap with Integers mapping to Future results
private final ConcurrentHashMap<Integer, Future<List<Integer>>> results
            = new ConcurrentHashMap<Integer, Future<List<Integer>>>();
public List<Integer> service (final int input) throws Exception {

  // the future result of factorization
  Future<List<Integer>> f;

  // set up the futuretask
  Callable<List<Integer>> eval = new Callable<List<Integer>>() {
      public List<Integer> call () throws InterruptedException {
          return factor(input);
      }
  };
  FutureTask<List<Integer>> ft = new FutureTask<List<Integer>>(eval);

  // execute put if absent, if there was no mapping, will put and
  // return null. Use f for convenience
  f = results.putIfAbsent(input, ft);

  // If there was no previous mapping, go execute the FutureTask.
  if (f == null) {
      f = ft;
    ft.run();
  }
  return f.get();
}
```

CopyOnWriteArrayList Use this when you must have **many threads iterating over the ArrayList**. If you don't need to be iterating over long lists with many threads, there is no need for it. This is a thread-safe variant of ArrayList in which all mutative operations (add, set, and so on) are implemented by making a fresh copy of the underlying array. This is ordinarily too costly, but may

be more efficient than alternatives when traversal operations vastly outnumber mutations, and is useful when you cannot or don't want to synchronize traversals, yet need to preclude interference among concurrent threads. **This array never changes during the lifetime of the iterator, so interference is impossible** and the iterator is guaranteed not to throw ConcurrentModificationException. The iterator will not reflect additions, removals, or changes to the list since the iterator was created. All write operations are protected by the same lock and read operations are not protected.

BlockingQueue (Interface) A BlockingQueue is typically used to have on thread produce objects, which another thread consumes. BlockingQueue uses add() and remove() (which attempt to insert/remove immediately and throw Exceptions if unable), or offer() and poll() (which attempt to insert/remove immediately and return false if unable), or put() and take() (which attempt to insert/remove immediately, and wait/block if necessary till elements/spaces become available). Here is a diagram that illustrates this principle:



The producing thread will keep producing new objects and insert them into the queue, until the queue **reaches some upper bound** on what it can contain. If the blocking queue reaches its upper limit, the **producing thread is blocked** while trying to insert the new object. It remains blocked until a consuming thread takes an object out of the queue. The consuming thread keeps taking objects out of the blocking queue, and processes them. If the consuming thread tries to take an object out of an empty queue, the **consuming thread is blocked** until a producing thread puts an object into the queue. An example:

```
class Producer implements Runnable {
 private final BlockingQueue queue;
 Producer(BlockingQueue q) { queue = q; }
 public void run() {
   try {
     while (true) { queue.put(produce()); }
   } catch (InterruptedException ex) { ... handle ...}
 }
 Object produce() { ... }
}

class Consumer implements Runnable {
 private final BlockingQueue queue;
 Consumer(BlockingQueue q) { queue = q; }
 public void run() {
   try {
     while (true) { consume(queue.take()); }
   } catch (InterruptedException ex) { ... handle ...}
 }
 void consume(Object x) { ... }
}

class Setup {
 void main() {
```

```
   BlockingQueue q = new SomeQueueImplementation();
   Producer p = new Producer(q);
   Consumer c1 = new Consumer(q);
   Consumer c2 = new Consumer(q);
   new Thread(p).start();
   new Thread(c1).start();
   new Thread(c2).start();
 }
}
```

FutureTask FutureTask constructors can accept either Runnable or Callable. Runnable and Callable interfaces are designed to represent thread-executable tasks. Why use Callable? A Callable needs to implement call() method while a Runnable needs to implement run() method. A Callable can return a value but a Runnable cannot. A Callable can throw checked exception but a Runnable cannot. FutureTasks are a cancellable asynchronous computation.

This class provides a base implementation of Future, with methods to start and cancel a computation, query to see if the computation is complete, and retrieve the result of the computation. The result can only be retrieved when the computation has completed; the get methods will block if the computation has not yet completed. Once the computation has completed, the computation cannot be restarted or cancelled (unless the computation is invoked using runAndReset()). A FutureTask can be used to wrap a Callable or Runnable object. **Because FutureTask implements Runnable, a FutureTask can be submitted to an Executor (or Thread!) for execution**. In addition to serving as a standalone class, this class provides protected functionality that may be useful when creating customized task classes. Methods are as follows:

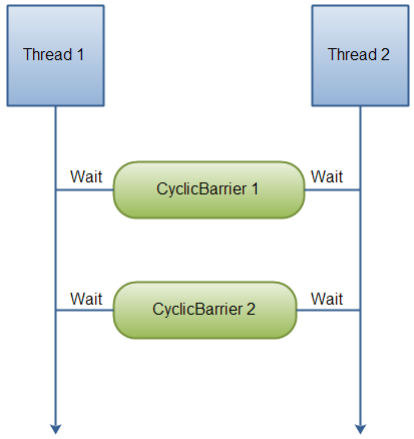
- `boolean cancel(boolean mayInterruptIfRunning)` Attempts to cancel execution of this task.
- `protected void done()` Protected method invoked when this task transitions to state isDone (whether normally or via cancellation).
- `V get()` Waits if necessary for the computation to complete, and then retrieves its result.
- `V get(long timeout, TimeUnit unit)` Waits if necessary for at most the given time for the computation to complete, and then retrieves its result, if available.
- `boolean isCancelled()` Returns true if this task was cancelled before it completed normally.
- `boolean isDone()` Returns true if this task completed.
- `void run()` Sets this Future to the result of its computation unless it has been cancelled.
- `protected boolean runAndReset()` Executes the computation without setting its result, and then resets this future to initial state, failing to do so if the computation encounters an exception or is cancelled.
- `protected void set(V v)` Sets the result of this future to the given value unless this future has already been set or has been cancelled.

6.4.3 Synchronizers

A synchronizer is an object that coordinates the control flow of threads based on its state.

Semaphore Semaphores are counters that increment/decrement through acquire() and release(). A binary semaphore acts as a mutex. A counting semaphore acts as a set of N permits.

CyclicBarrier The java.util.concurrent.CyclicBarrier class is a synchronization mechanism that can synchronize threads progressing through some algorithm. In other words, it is a barrier that all threads must wait at, until all threads reach it, before any of the threads can continue. The threads wait for each other by calling the await() method on the CyclicBarrier. Once N threads are waiting at the CyclicBarrier, all threads are released and can continue running. Here is a diagram illustrating that:



If a timeout is required, `barrier.await(N, TimeUnit.SECONDS)` can be used. A thread is released after N seconds even if not all threads have reached the Barrier. The waiting threads waits at the CyclicBarrier until either:

- The last thread arrives (calls await()).
- The thread is interrupted by another thread (another thread calls its interrupt() method).
- Another waiting thread is interrupted.
- Another waiting thread times out while waiting at the CyclicBarrier.
- The `CyclicBarrier.reset()` method is called by some external thread.

CountDownLatch CountDownLatch is a concurrency construct that allows one or more threads to wait for a given set of operations to complete. A CountDownLatch is initialized with a given count. This count is decremented by calls to the countDown() method. Threads waiting for this count to reach zero can call one of the await() methods. Calling await() blocks the thread until the count reaches zero. Below is a simple example. After the Decrementer has called countDown() 3 times on the CountDownLatch, the waiting Waiter is released from the await() call.

```
import java.util.concurrent.CountDownLatch;

public class CDLDemo {

    public static void main(String[] args) throws Exception {
        CountDownLatch latch = new CountDownLatch(3);

        Waiter waiter = new Waiter(latch);
        Decrementer decrementer = new Decrementer(latch);

        new Thread(waiter).start();
        new Thread(decrementer).start();
    }
}

class Waiter implements Runnable {
```

```
    CountDownLatch latch = null;

    public Waiter(CountDownLatch latch) {
        this.latch = latch;
    }

    public void run() {
        try {
            latch.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Waiter Released");
    }
}

class Decrementer implements Runnable {

    CountDownLatch latch = null;

    public Decrementer(CountDownLatch latch) {
        this.latch = latch;
    }

    public void run() {
        try {
            Thread.sleep(1000);
            this.latch.countDown();
            System.out.println(latch.getCount());

            Thread.sleep(1000);
            this.latch.countDown();
            System.out.println(latch.getCount());
            Thread.sleep(1000);
            this.latch.countDown();
            System.out.println(latch.getCount());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

// output:
// 2
// 1
// 0
// Waiter Released
```

Phaser When to use Phaser (as compared to CyclicBarrier or CountDownLatch): When you don't know the number of threads before run time. That is, when the number of threads vary depending on run time. To give an analogy, consider an university admission: you can have 1000 students register in one year and maybe 800 in another. You only know how many students register when you open the admission portal. CyclicBarrier and CountDownLatch is more suited for applications when you know the number of registered threads before run time. An analogy would be an examination setting where the number of exam takers is known before the exam. What is arriving and waiting (methods arrive(), awaitAdvance() and arriveAndAwaitAdvance())? The notion of arrival is separate from the notion of waiting. Consider this analogy of a theme park ride: after you registered for the ride, you are given a queue number after arrival. You can play other rides before your queue number starts. You eventually return to the first ride and await the ride to start.

- **Registration**. Unlike the case for other barriers, the number of parties registered to synchronize

on a phaser may vary over time. Tasks may be registered at any time (using methods register(), bulkRegister(int), or forms of constructors establishing initial numbers of parties), and optionally deregistered upon any arrival (using arriveAndDeregister()). As is the case with most basic synchronization constructs, registration and deregistration affect only internal counts; they do not establish any further internal bookkeeping, so tasks cannot query whether they are registered. (However, you can introduce such bookkeeping by subclassing this class.)

- **Synchronization**. Like a CyclicBarrier, a Phaser may be repeatedly awaited. Method arriveAndAwaitAdvance() has effect analogous to CyclicBarrier.await. Each generation of a phaser has an associated phase number. The phase number starts at zero, and advances when all parties arrive at the phaser, wrapping around to zero after reaching Integer.MAX_VALUE. The use of phase numbers enables independent control of actions upon arrival at a phaser and upon awaiting others, via two kinds of methods that may be invoked by any registered party:

- **Arrival**. Methods arrive() and arriveAndDeregister() record arrival. These methods do not block, but return an associated arrival phase number; that is, the phase number of the phaser to which the arrival applied. When the final party for a given phase arrives, an optional action is performed and the phase advances. These actions are performed by the party triggering a phase advance, and are arranged by overriding method onAdvance(int, int), which also controls termination. Overriding this method is similar to, but more flexible than, providing a barrier action to a CyclicBarrier.

- **Waiting**. Method awaitAdvance(int) requires an argument indicating an arrival phase number, and returns when the phaser advances to (or is already at) a different phase. Unlike similar constructions using CyclicBarrier, method awaitAdvance continues to wait even if the waiting thread is interrupted. Interruptible and timeout versions are also available, but exceptions encountered while tasks wait interruptibly or with timeout do not change the state of the phaser. If necessary, you can perform any associated recovery within handlers of those exceptions, often after invoking forceTermination. Phasers may also be used by tasks executing in a ForkJoinPool, which will ensure sufficient parallelism to execute tasks when others are blocked waiting for a phase to advance.

- **Termination**. A phaser may enter a termination state, that may be checked using method isTerminated(). Upon termination, all synchronization methods immediately return without waiting for advance, as indicated by a negative return value. Similarly, attempts to register upon termination have no effect. Termination is triggered when an invocation of onAdvance returns true. The default implementation returns true if a deregistration has caused the number of registered parties to become zero. As illustrated below, when phasers control actions with a fixed number of iterations, it is often convenient to override this method to cause termination when the current phase number reaches a threshold. Method forceTermination() is also available to abruptly release waiting threads and allow them to terminate.

- **Tiering**. Phasers may be tiered (i.e., constructed in tree structures) to reduce contention. Phasers with large numbers of parties that would otherwise experience heavy synchronization contention costs may instead be set up so that groups of sub-phasers share a common parent. This may greatly increase throughput even though it incurs greater per-operation overhead. In a tree of tiered phasers, registration and deregistration of child phasers with their parent are managed automatically. Whenever the number of registered parties of a child phaser becomes non-zero (as established in the Phaser(Phaser,int) constructor, register(), or bulkRegister(int)), the child phaser is registered with its parent. Whenever the number of registered parties becomes zero as the result of an invocation of arriveAndDeregister(), the child phaser is deregistered from its parent.

- **Monitoring**. While synchronization methods may be invoked only by registered parties, the current state of a phaser may be monitored by any caller. At any given moment there are getRegisteredParties() parties in total, of which getArrivedParties() have arrived at the current phase (getPhase()). When the remaining (getUnarrivedParties()) parties arrive, the phase advances.

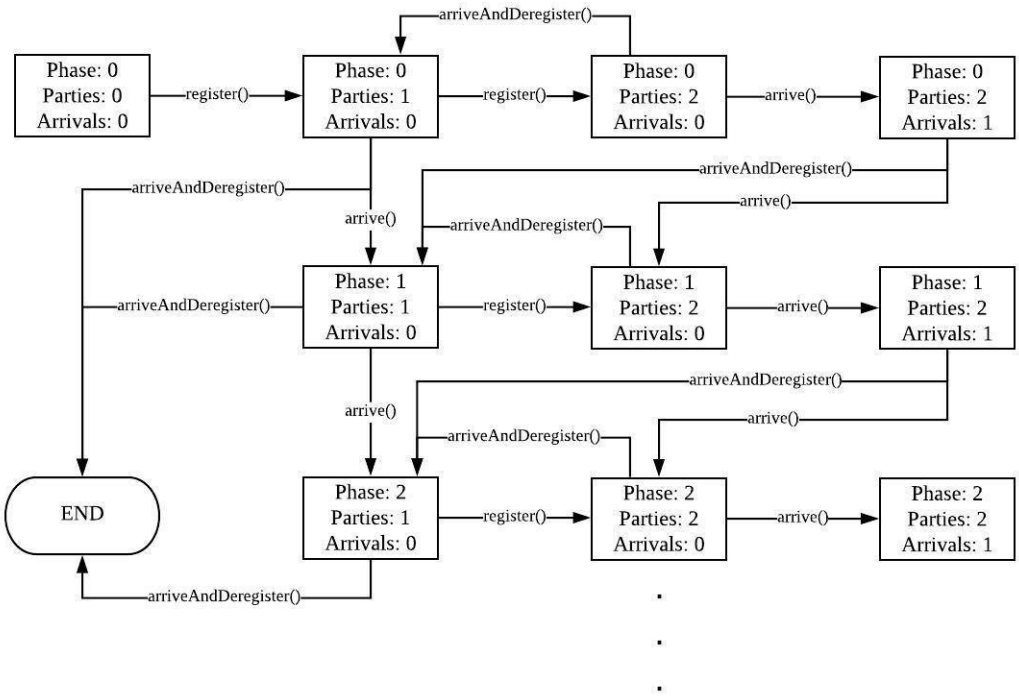
The values returned by these methods may reflect transient states and so are not in general useful for synchronization control. Method toString() returns snapshots of these state queries in a form convenient for informal monitoring. This also differs from CyclicBarrier or CountDownLatch where both arriving and waiting is combined.

Using `Phaser` to dynamically wait for N number of threads:

```
void runTasks(List<Runnable> tasks) {
  final Phaser phaser = new Phaser(1); // "1" to register self
  // create and start threads
  for (final Runnable task : tasks) {
   phaser.register();
   new Thread() {
     public void run() {
       phaser.arriveAndAwaitAdvance(); // await all creation
       task.run();
     }
   }.start();
  }

  // allow threads to start and deregister self
  phaser.arriveAndDeregister();
}
```

Yap Wei Lok 1002394

Since arrive() and arriveAndAwaitAdvance() is fairly similar in terms of state transitions, I will only present arrive()



7 Testing Concurrent Programs

1. Create an object of the class, call its methods (in different sequences with different inputs) and assert post-conditions and invariants.
2. Set up multiple threads performing operations over some amount of time and then somehow test that nothing went wrong. Mind that the test programs are concurrent programs too! It's best if checking the test property does not require any synchronization.
3. How do we test that everything put into the buffer comes out of it and that nothing else does, assuming there are multiple producers and consumers?
   (a) A naïve approach: maintain a "shadow" list and assert that the buffer is consistent with the "shadow" list
   (b) Use a checksum function would be better
4. Some test data should be generated randomly.
5. Random number generator can create couplings between classes and timing artifacts because most random number generator classes are thread-safe and therefore introduce additional synchronization.
6. Often we need to generate more scheduling, especially if the time to completion is small. We can do things like:
   (a) Test with more active threads than CPUs.
   (b) Testing with different processor counts, operating systems, and processor architectures.
   (c) Encourage context switching using Thread.yield() or Thread.sleep.
7. Or just use tools (Team Explorer, Codeclimate, Coverity Scan, ...).

8 Optimization in Concurrency

8.1 Assigning Threads for Tasks

8.1.1 One thread per task

The idea is to create a new thread for every task that we need to execute. This is responsive, but you end up with a huge number of threads in real-world applications (e.g. large web servers). You gain parallel processing, but the code has to be thread-safe. Susceptible to DDoS because of unbound thread creation. Thread creation and tear down involves the JVM and OS. For lots of lightweight threads this is not very efficient. Active Threads consume extra memory, for instance to provide for a thread stack. If there are less CPU's than threads, some threads sit idle, consuming memory. There is a limit on how many threads you can have concurrently. If you hit this limit your program will most likely become unstable.

8.2 Thread Pooling

Using worker threads minimizes the overhead due to thread creation. Thread objects use a significant amount of memory, and in a large-scale application, allocating and deallocating many thread objects creates a significant memory management overhead. One common type of thread pool is the fixed thread pool. This type of pool always has a specified number of threads running; if a thread is somehow terminated while it is still in use, it is automatically replaced with a new thread. Tasks are submitted to the pool via an internal queue, which holds extra tasks whenever there are more active tasks than threads. By properly tuning the size of the thread pool, you can have enough threads to keep the

processors busy while not having so many that your application runs out of memory or thrashes (screws up the garbage collector when a large number of temporary objects are created in very short intervals) due to competition among threads for resources.

8.2.1 Executor and ExecutorService

Delegation. The **Executor** (Interface) framework offers flexible thread pool management, provides a standard means of decoupling task submission from task execution. Executor just executes stuff you give it, while ExecutorService adds startup, shutdown, and the ability to wait for and look at the status of jobs you've submitted for execution on top of Executor (which it extends); ExecutorService offers **submit(), invoke() and shutdown() functions**. Using shutdown() will just tell the executor service that it can't accept new tasks, but the already submitted tasks continue to run. There's also shutdownNow() which will do the same AND will try to cancel the already submitted tasks by interrupting the relevant threads. Note that if your tasks ignore the interruption, shutdownNow() will behave exactly the same way as shutdown().

8.2.2 Thread pool types

- **newFixedThreadPool**, Fixed-size thread pool; creates threads as tasks are submitted, up to the maximum pool size and then attempts to keep the pool size constant
- **newCachedThreadPool**, Boundless, but the pool shrinks and grows when demand dictates so
- **newSingleThreadExecutor**, A single worker thread to process tasks, sequentially according to the order imposed by the task queue
- **newScheduledThreadPool**, A fixed-size thread pool that supports delayed and periodic task execution.

8.2.3 Execution policy and Optimizing the thread pool

So when do I thread pool? Thread pools work best when **tasks are homogeneous and independent** (Dependency between tasks in the pool creates constraints on the execution policy which might result in deadlock, liveness, etc.).

The optimal pool size, S_{pool} is:

$$S_{pool} = N_{CPU} \cdot u_{CPU} \cdot \left(1 + \frac{t_{wait}}{t_{compute}}\right) \tag{1}$$

Where N_{CPU} is the number of CPUs, u_{CPU} is the target CPU utilization ϵ $(0, 1)$, and $\frac{t_{wait}}{t_{compute}}$ is the ratio of waiting time to compute time. Decoupling submission from execution is that it lets you specify the execution policy for a given class of tasks:

- In what thread will tasks be executed?
- In what order should tasks be executed (FIFO)?
- How many tasks may execute concurrently?
- How many tasks may be queued pending execution?
- If a task has to be rejected because the system is overloaded, which task should be selected and how the application be notified?
- What actions should be taken before or after executing a task?

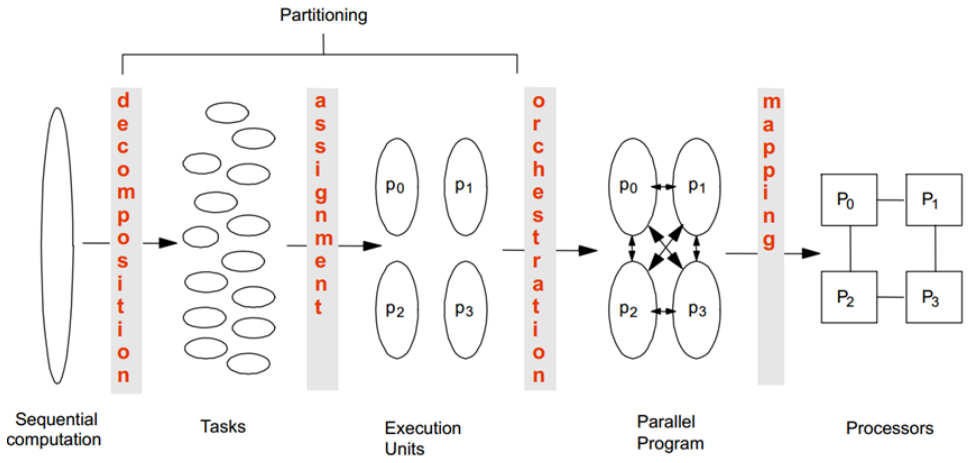
Alternative ways of tuning Other resources that can contribute to sizing constraints are memory, file handles, socket handles, database connections, etc. Add up how much of those resources each task requires and divide that into the total quantity available. Alternatively, the size of the thread pool can be tuned by running the application using different pool sizes and observing the level of CPU and other resource utilization.

Writing the code The execution policy is **defined in the execute() method of your concrete Executor class**. Typically, the Executor is set up in this fashion:

1. `private static final int NTHREADS = 50;`
2. `private static final Executor exec = Executors.newFixedThreadPool(NTHREADS);`
3. `Runnable task = new Runnable () { public void run() { //something }};`
4. `exec.execute(task);`

8.3 Parallelization

8.3.1 How to maximize performance by defining tasks?



1. Decomposition: Break up the computation into "self-contained" tasks to be divided among processes. Tasks should be medium-sized. If too small, the ratio of useful work vs overhead becomes small. If too big, the number of tasks available at a time is upper bound on achievable speedup. Tasks may become available dynamically!
2. Assignment: Specify mechanism to divide work among cores. We may want to balance the amount of work for each core and reduce communication between the threads and apply Well-known design patterns.
3. Orchestration and Mapping: Figure out what kind of communication is needed between each pair of threads. Less communication is better: preserve locality of data! Schedule the threads to satisfy tasks' dependencies and use Executor to manage the thread pool.

A note on dependency Dependency is defined, by Bernstein's condition, if two tasks cannot run in parallel because of some action or output that preceeds the input of the next task.

What if you don't want to use Executor? Then we need to consider the number of execution units platform will support (we have a maximum number of threads!), and the cost of sharing information among execution units. In this case, optimize to do things like DFS in parallel.

8.4 Design Patterns for Parallelization

8.4.1 Pattern 1: Single Program Multiple Data (SPMD)

All threads/processes run the same program, operating on different data. This model is particularly appropriate for problems with a regular, predictable communication pattern. MATLAB supports SPMD blocks. Often adopted for GPU programming. In the cohort exercise we split an integration task by assigning threads for different intervals.

8.4.2 Pattern 2: Loop Parallelism

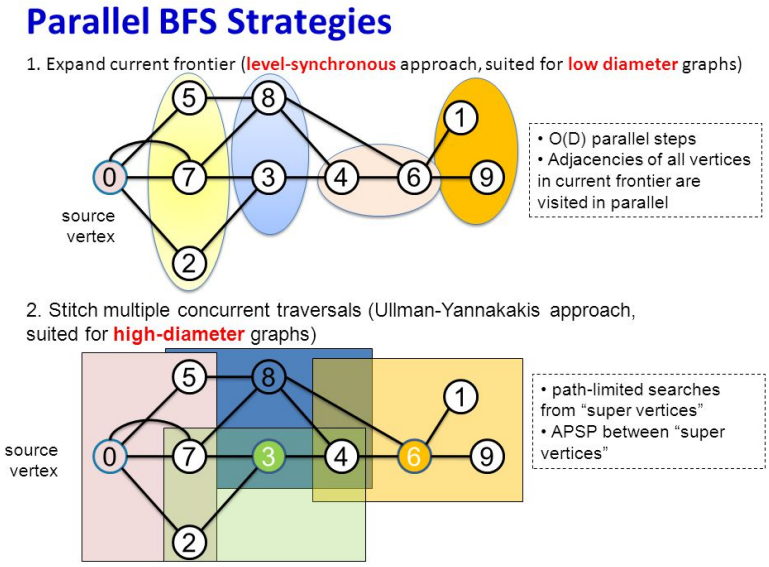
If we need to loop from $i = 0$ to $i < 3000$, Thread A can loop from 0 to 999, B from 1000 to 1999, C from 2000 to 2999.

8.4.3 Pattern 3: Master-Worker Pattern

A master thread/process divides a problem into several sub-problems and dispatches them to several worker processes.

8.4.4 Pattern 4: Fork-Join Pattern

Tasks can create more tasks! Tasks manage other tasks according to their relationship. Parent task creates new tasks (fork) then waits until they complete (join) before continuing on with the computation. For example, **How to parallelize BFS?** Nodes of the same hop-distance (level) from the root can be visited in parallel by child threads:



8.5 Dealing with Limitations in Concurrency

8.5.1 Limits: Amdahl's law

Amdahl's law says that on a machine with N processors, and F is the fraction of the code that must be executed serially we can achieve a speed up of at most:

$$speedup \leq \frac{1}{F + \frac{1-F}{N}} \tag{2}$$

All concurrent programs have some degree of **serialization**.

8.5.2 The cost of using threads

Context switching requires saving the context of the currently running thread and restoring the execution context of the newly scheduled thread. **Cache misses** are deadly, cost about 5,000 to 10,000 clock cycles. In memory synchronization, memory barriers inhibit compiler optimization.

8.5.3 The cost of using locks

Locks naturally induce serialization. A thread with the lock may be delayed (due to a page fault, scheduling delay, etc.). One thread at a time can acquire locks and execute critical section code. Performance takes a hit if many threads are vying for the lock, or the locks are being held for long periods of time. Locking is simply a heavyweight mechanism for simple operations like count++. So how do we fix this?

```
public synchronized boolean userLocationMatches (String name, String regexp) {
  String key = "users." + name + ".location";
  String location = attributes.get(key);

  if (location == null) {
    return false;
  }
  else {
    return Pattern.matches(regexp, location);
  }
}
```

Refactor! Here, the entire method is synchronized, leading to serialization when there are many threads trying to get the key. But we notice that the rest of the method, other than `attributes.get()` **doesn't** need to be synchronized! We can effectively reduce the serialization to:

```
public boolean userLocationMatches (String name, String regexp) {
  String key = "users." + name + ".location";
  String location;
  synchronized(this) {
    location = attributes.get(key)
  }
  if (location == null) {
    return false;
  }
  else {
    return Pattern.matches(regexp, location);
  }
}
```

Or even better, delegate concurrency to ConcurrentHashMap. We assume there exists a new `private final Map<String, String> attributes = new ConcurrentHashMap<String, String>()`, eliminating the need for manual locking:

```
public boolean userLocationMatches (String name, String regexp) {
  String key = "users." + name + ".location";
  String location = attributes.get(key); // the get method in CHM is already thread-safe
```

```
  if (location == null) {
    return false;
  }
  else {
    return Pattern.matches(regexp, location);
  }
}
```

In general, we can improve performance by **locking on different objects** to reduce the time and demand for locks. But of course these have to make sense and must still be thread-safe. We can also do **lock striping** like in the case of `ConcurrentHashMap`. Even more alternatives exist in the form of **read-write locks** (more than one reader can access the shared resource concurrently, but writers must acquire the lock exclusively), **immutable objects** and **atomic variables**.

8.5.4 Using Future

Place time limits on tasks by using `Future.get(long timeout, TimeUnit unit)` to time out.

```
public class FutureRenderer2 {
  private final ExecutorService executor = new ScheduledThreadPoolExecutor (100);

  void renderPage (CharSequence source) throws Exception {
    final List<ImageInfo> imageInfos = scanForImageInfo(source);

    Callable<List<ImageData>> task = new Callable<List<ImageData>>() {
      public List<ImageData> call () {
        List<ImageData> result = new ArrayList<ImageData>();
        for (ImageInfo imageInfo : imageInfos) {
          result.add(imageInfo.downloadImage());
        }

        return result;
      }
    };

    Future<List<ImageData>> future = executor.submit(task);
    renderText(source);

    try {
      List<ImageData> imageData = future.get(3, TimeUnit.SECONDS);
      for (ImageData data: imageData) {
        renderImage(data);
      }
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
      future.cancel(true);
    } catch (TimeoutException e) {
      renderCross();
    }
  }
  private void renderCross() { /* TODO Auto-generated method stub */ }
  private void renderImage(ImageData data) { /* TODO Auto-generated method stub */ }
  private void renderText(CharSequence source) { /* TODO Auto-generated method stub */ }
  private List<ImageInfo> scanForImageInfo(CharSequence source) { return null; }
}
```

8.6 Non-Blocking Synchronization

Very simplified, the idea is to keep checking if some other thread has already modified your stuff in an atomic way. An algorithm is called non-blocking if failure or suspension of any thread cannot cause failure or suspension of another thread. Non-blocking algorithms are immune to deadlock (though, in unlikely scenarios, may exhibit livelock or starvation) Examples: Stacks (Treiber's), queues, hash tables, etc.

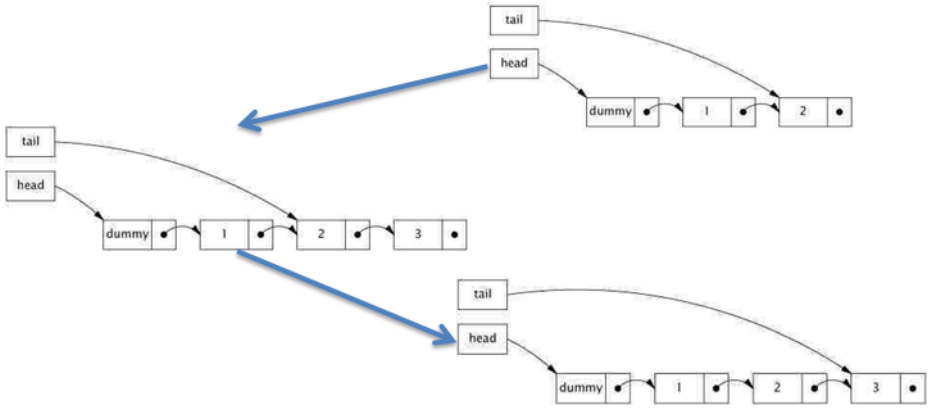
8.6.1 Hardware support

Compare-And-Swap CAS has three operands: a memory location V, the expected old value A, and the new value B. CAS updates V to the new value B, but only if the value in V matches the expected old value A; otherwise, it does nothing. In either case, it returns the value currently in V. CAS is supported in atomic variable classes.

Example: Non-blocking stack Java has `AtomicReference`, an object reference that may be updated atomically. It has lots of atomic methods like getAndSet(), compareAndSet(), and accumulateAndGet().

```
public class ConcurrentStack <E> {
  AtomicReference<Node<E>> top = new AtomicReference<Node<E>>();

  public void push(E item) {
    Node<E> newHead = new Node<E>(item);
    Node<E> oldHead;
    do {
      oldHead = top.get();
      newHead.next = oldHead;
    } while (!top.compareAndSet(oldHead, newHead)); //commitment here
  }

  public E pop() {
    Node<E> oldHead;
    Node<E> newHead;
    do {
      oldHead = top.get();
      if (oldHead == null)
        return null;
      newHead = oldHead.next;
    } while (!top.compareAndSet(oldHead, newHead)); //commitment here
    return oldHead.item;
  }

  private static class Node <E> {
    public final E item;
    public Node<E> next;

    public Node(E item) {
      this.item = item;
    }
  }
}
```

Example: Non-blocking linked-list



If the queue is in the normal state, tail.next is null. In that case, then we can try and modify it. All pointers are AtomicReferences.

```
public boolean put(E item) {
  Node<E> newNode = new Node<E>(item, null);
  while (true) {
    Node<E> curTail = tail.get();
    Node<E> tailNext = curTail.next.get();
    if (curTail == tail.get()) { //if this is not true, abort since other some other thread has already modified
tail;
      if (tailNext != null) { // if this is true, Queue in intermediate state, advance tail
        tail.compareAndSet(curTail, tailNext);
      } else {
        // In normal state, try inserting new node
        if (curTail.next.compareAndSet(null, newNode)) {
          // Insertion succeeded, try advancing tail
          tail.compareAndSet(curTail, newNode);
          return true;
        }
      }
    }
  }
}
```