

**07/10/1399**

مهندسی نرم افزار 2



تمرین پنجم



**گروه 2**

**اعضاء گروه:**

1) محمدرضا اخگری (9631001)

2) محمدعلی کشاورز (9631061)

3) علی نظری (9631075)

1. درباره رهنمودهای ساخت نرم افزار قابل نگهداری و معیارهای ارزیابی قابلیت نگهداری، که توسط Software Improvement Group (SIG) معرفی شده است، مطالعه کنید[[1]](#footnote-1).

الف) رهنمودهای مطرح شده را مختصراً معرفی کرده و (بر اساس نظر و تجربه خود) اولویت بندی کنید.

ب) قابلیت نگهداری یک پروژه دلخواه را با استفاده از روش پیشنهاد شده این گروه بررسی کنید. نتایج به دست آمده را تحلیل کنید.

الف)

**رهنمودهای ساخت نرم افزار قابل نگهداری:**

* **بخش‌های کد را کوتاه نگهدار**: حداکثر طول کد هر یونیت را 15 خط نگهدار و این کار را با شکستن واحد های بزرگ انجام بده چون باعث میشه که خوانایی و فهم کد بیشتر بشه.
* **بخش‌های کد را ساده بنویس**: تعداد برنچ ها را درون هر یونیت از کد نهایتا به 4 محدود کنیم چون هر چه تعداد برنچ ها کمتر باشد امکان نگهداری محصول بیشتر است.
* **کد را یک بار بنویس**: کد رو به هیچ وجه نباید کپی کرد، یا باید یه کد جنریک نوشت یا یک فانکشن نوشت و کال کرد. این باعث میشه که باگ رو مجبور نباشیم چند جا فیکس کنیم.
* **اینترفیس های بخش های مختلف کد را کوچک نگهدار**: این یعنی پارامتر های مربوط به هر یونیت رو به 4 تا محدود کنیم و این کار رو با استفاده از آبجکت ها به عنوان پارامتر انجام دهیم.
* **نگرانی ها و مسئولیت ها را در ماژول ها جدا نگهدار**: برای کم کردن میزان وابستگی بین ماژول ها، ماژول ها بزرگ نسازید این کار باعث میشود مسئولیت های مختلف توسط یک ماژول انجام شود؛ بلکه میبایست وابستگی بین ماژول ها را با جدا کردن مسئولیت ها کم کنیم.
* **وابستگی ماژول ها رو کم کن**: این کار را با حداقل کردن کدها relative در ماژول ها انجام دهیم، این موضوع باعث میشود که نگهداری از کد راحت تر شود چون اجزای مستقل نگهداری ایزوله ای دارند.
* **بالانس را در طراحی کامپوننت ها حفظ کن**: به این معناست که تعداد و اندازه نسبی اجزای تاپ لول را متعادل نگهداریم، یعنی تعداد را بین 6 تا 12 نگهداریم و اندازه اجزا به نسبت یکدیگر تقریبا برابر باشد.
* **کد را کوچک نگه دار**: کد خود را تا حد امکان کوچک نگهدایم. این کار را بایستی با کاهش فعال اندازه کد انجام دهیم.
* **تست های خودکار بنویس**: برای کد خود تست های خودکار بنویسیم این موضوع باعث میشود که با هر بار تغییر کد بتوان تست هارا اجرا کرد و تغییر را کم خطرتر میکند.
* **کد تمیز بنویس**: با جلوگیری از استفاده از code smells بعد از پیاده سازی میتوانیم قابلیت نگهداری را در محصول خودمان افزایش دهیم.

ب) برای هر پروژه‌ای می‌توان این معیارها را در نظر گرفت ولی ما یک سری کد متن‌باز را درنظر گرفتیم که در هر مورد یک مثال زده‌ایم و باقی قسمت‌ها را توضیح دادیم، برای همین برای هر بخش تکه کدی را در نظر گرفتیم و این مسائل را اعمال کردیم.

* Write short units of code
* Write simple units of code
* Write code once
* Keep unit interfaces small
* Separate concerns in modules
* Couple architecture components loosely
* Keep architecture components balanced
* Keep your codebase small
* Automate tests
* Write clean code

**مورد اول:**

اولین مورد نشان می‌دهد که توابع ما باید کوتاه باشند، معمولاً بیش از 15 خط کد نیست. این امر نه تنها خوانایی را بهبود می‌بخشد (فهم خط کد کمتر)، بلکه، یک تابع کوتاه تغییرات کمتری خواهد داشت، به این معنی که تست آن آسان‌تر است.

آسانترین راه برای استفاده از این روش انتقال قسمت‌هایی از کد در یک تابع به تابع‌های دیگر است. بسیاری از IDEها عملکرد استخراج تابع را دارند که این کار را آسان‌تر می‌کند. با این حال، گاهی اوقات، پاسخ صحیح انتقال کد نه به تابع دیگری، بلکه به یک کلاس جدید است - وقتی به رهنمودهای معماری می‌رسیم، موارد بیشتری از آن را خواهیم دید.

برای شمارش تعداد خط، ممکن است در تیم‌های مختلف معیار متفاوتی قرار داده شود ولی ما از موارد زیر استفاده می‌کنیم:

* خط تعریف و براکت‌های باز و بسته شدن تابع حساب نمی‌شود. به این دلیل که اینها خطوطی هستند که قابل حذف نیستند و بنابراین هیچ تأثیری در اندازه‌گیری پیچیدگی تابع ندارند.
* خطوط خالی درون تابع شمارش می‌شوند. این به این دلیل است که، اگرچه خطوط خالی هیچ دستورالعملی ندارند، برنامه‌نویسان تمایل دارند که آنها را به انتهای خط‌هایی اضافه کنند که کارهای نزدیک بهم را انجام می‌دهند (یک گروهند)، به این معنی که آنها به پیچیدگی تابع کمک می‌کنند.
* اگر یک دستورالعمل آنقدر طولانی است که باید به دو یا چند خط تقسیم شود، هر یک از آن سطرها را به طور مستقل می‌شماریم. این بدان دلیل است که ما چنین دستورالعمل‌هایی را به عنوان پیچیدگی اضافی در نظر می‌گیریم و بنابراین منطقی است که آنها بیشتر به تعداد کل خط کمک کنند.

بیایید نگاهی به یک مثال بیندازیم. تابع زیر شامل 21 خط کد، بیش از حد توصیه شده است (15 خط). ممکن است مشخص نباشد که این تابع خاص چه کاری انجام می‌دهد، اما در این مرحله مهم نیست (این بخشی از ابزاری برای تجزیه و تحلیل داده‌های ساخت از یک سرور جنکینز است)

protected void selectBuilds(String source) {

    jenkinsClient = new JenkinsClient(source);

    List<String> allBuilds = jenkinsClient.getBuildConfigurations();

    BuildSelector buildSelector = new BuildSelector(allBuilds);

    GridPane grid = new GridPane();

    grid.setAlignment(Pos.CENTER);

    grid.setHgap(10);

    grid.setVgap(10);

    grid.setPadding(new Insets(25, 25, 25, 25));

    grid.add(buildSelector, 0, 0);

    Scene scene = new Scene(grid, 250, 400);

    m\_primaryStage.setScene(scene);

    Button btn = new Button();

    btn.setText("Show me hotspots!");

    btn.setOnAction(event -> {

        List<String> selectedBuilds = buildSelector.getBuilds();

        AddDrawingToScene(selectedBuilds);

    });

    grid.add(btn, 0, 1);

}

ما می‌توانیم این روش را با انتقال create و setup اشیا Grid و Button به تابع‌های جدید کوتاه‌تر کنیم ، مانند این:

private GridPane createGridPane() {

    GridPane grid = new GridPane();

    grid.setAlignment(Pos.CENTER);

    grid.setHgap(10);

    grid.setVgap(10);

    grid.setPadding(new Insets(25, 25, 25, 25));

    return grid;

}

private Button createButton(BuildSelector buildSelector) {

    Button btn = new Button();

    btn.setText("Show me hotspots!");

    btn.setOnAction(event -> {

        List<String> selectedBuilds = buildSelector.getBuilds();

        AddDrawingToScene(selectedBuilds);

    });

    return btn;

}

protected void selectBuilds(String source) {

    jenkinsClient = new JenkinsClient(source);

    List<String> allBuilds = jenkinsClient.getBuildConfigurations();

    BuildSelector buildSelector = new BuildSelector(allBuilds);

    GridPane grid = createGridPane();

    grid.add(buildSelector, 0, 0);

    Scene scene = new Scene(grid, 250, 400);

    m\_primaryStage.setScene(scene);

    Button btn = createButton(buildSelector);

    grid.add(btn, 0, 1);

}

اکنون ما سه تابع کوچک در مقابل یک تابع بزرگ داریم که دستکاری کد را آسان می‌کند.

**مورد دوم:**

هرچه راه‌های اجرای یک تابع بیشتر باشد، استدلال در مورد همه آن‌ها دشوارتر خواهد بود و وقتی استدلال در مورد کد مشکل باشد، سوتفاهم رخ می‌دهد و سوتفاهم منجر به باگ می‌شود.

مهم است که روشن شود معنای مسیر اجرای کد چیست. مسیرهای اجرا نقاط منشعب هستند، دستورالعمل‌هایی که می‌توانند اجرای کد را به طریقی یا دیگری انجام دهند. به عنوان مثال، دستور if شاخه‌ای از اجرا را ایجاد می‌کند زیرا بسته به ارزیابی یک شرط، کد متفاوتی اجرا می‌شود.

این راهنما پیشنهاد می‌کند که ما نقاط شاخه را حداکثر به چهار محدود کنیم. این امر نه تنها درک توابع را آسان می‌کند بلکه تست آنها را نیز آسان می‌کند. برای اینکه همه سناریوهای مختلف یک تابع را پوشش دهیم، به تعدادی تست خودکار نیاز داریم که حداقل تعداد نقاط شاخه بعلاوه یک باشد. برای مثال کد زیر را ببینیم:

public int getDiscount(String promoCode) {

    if(promoCode == null) {

        throw new IllegalArgumentException("promoCode");

    }

    promoCode = promoCode.trim();

    if(promoCode.length() < 5 || promoCode.length() > 8) {

        throw new IllegalArgumentException("promoCode");

    }

    if(expiredPromoCodes.containsKey(promoCode)) {

        throw new ExpiredPromoException(promoCode);

    }

    if(!availablePromoCodes.containsKey(promoCode)) {

        return 0;

    }

    return availablePromoCodes.get(promoCode);

}

این تابع تخفیف مناسبی را برای اعمال بسته به کد تبلیغاتی فراهم می‌کند و موارد استثنایی را در شرایط خاص ایجاد می‌کند. این کد دارای پنج نقطه انشعاب است، به این معنی که شش سناریو مختلف وجود دارد که باید هنگام تست در نظر گرفته شود: کد تبلیغاتی null، کوتاه یا خیلی طولانی، منقضی شدن تبلیغات؛ کد تبلیغاتی با هیچ تبلیغ تبلیغاتی موجود مطابقت ندارد (منقضی شده است یا نه) و کد تبلیغاتی با موفقیت اعمال شد.

ما می توانیم با انتقال منطق اعتبار سنجی به تابع خاص خود، مانند موارد زیر، تعداد نقاط انشعاب را در unit کاهش دهیم.

public boolean isPromoCodeValid(String promoCode) {

    if(promoCode == null) {

        return false;

    }

    promoCode = promoCode.trim();

    if(promoCode.length() < 5 || promoCode.length() > 8) {

        return false;

    }

    return true;

}

public int getDiscount(String promoCode) {

    if(!isPromoCodeValid(promoCode)) {

        throw new IllegalArgumentException("promoCode");

    }

    promoCode = promoCode.trim();

    if(expiredPromoCodes.containsKey(promoCode)) {

        throw new ExpiredPromoException(promoCode);

    }

    if(!availablePromoCodes.containsKey(promoCode)) {

        return 0;

    }

    return availablePromoCodes.get(promoCode);

}

با نسخه جدید، ما دو تابع با هر سه نقطه انشعاب داریم، به این معنی که برای پوشش دادن همه موارد به چهار تست برای هر تابع نیاز خواهیم داشت. ممکن است به نظر برسد که الان کارهای بیشتری برای انجام داریم، زیرا در مجموع هشت سناریو داریم که باید پوشش دهیم، در حالی که پیش از این فقط شش سناریو داشتیم. با این حال، تجزیه و تحلیل تلاش از این طریق می‌تواند فریبنده باشد. ما هشت سناریو نداریم که بتوانیم آنها را پوشش دهیم. ما هر کدام دو مجموعه از چهار سناریو داریم. این تمایز از آن جهت مهم است که در توسعه نرم افزار، تلاش با پیچیدگی به صورت خطی رشد نمی‌کند، بلکه به صورت تصاعدی رشد می‌کند. بنابراین، مدیریت دو مجموعه از چهار سناریو آسانتر از یک با شش است.

**مورد سوم:**

روش‌های زیادی برای اشاره به این امر وجود دارد (DRY, WET و ...) زیرا این یکی از قدرتمندترین منابع اشکالات است.

معمولاً اینگونه پیش می‌رود: یک برنامه‌نویس، شاید به دلیل محدودیت‌های زمانی، تصمیم می‌گیرد بخشی از کد را کپی و جای‌گذاری کند تا از آن در جای دیگری استفاده کند. مدتی بعد از آن، الزامی برای اصلاح آن قطعه کد از راه می‌رسد. برنامه‌نویسی که این کار را انتخاب می‌کند، به یاد نمی‌آورد یا متوجه نمی‌شود کدی که باید اصلاح شود در دو مکان مختلف وجود دارد، بنابراین برنامه‌نویس فقط تغییرات را در یکی از نسخه‌ها اعمال می‌کند، این گونه ما به یک باگ میخوریم: دو قسمت از سیستم که قرار است همان کار را انجام دهند دیگر یک کار را نمی‌کنند.

اما حتی اگر ما نسخه‌برداری را به خوبی مدیریت کنیم و از باگ‌ها جلوگیری کنیم، کد تکراری هنوز هم می‌تواند به یک تیم آسیب برساند. هر زمان که کاری انجام می‌شود، اگر برنامه نویسان بدانند که در کد موارد تکراری وجود دارد، باید تمام موارد کد را که باید اصلاح شوند جستجو کنند و به طور مناسب بر روی همه آنها عمل کنند. این هزینه بسیار بیشتری دارد که مجبور به تغییر فقط یک نسخه موجود از کد باشند.

نکته اصلی این است که ، هر زمان که کد تکراری را مشاهده کردید باید آن را به یک نسخه تبدیل کنید.

**مورد چهارم:**

توابع با تعریف طولانی معمولاً وجود داده‌های زیاد را نشان می‌دهند، مثاله‌ای نمونه‌ای از انبوه داده‌ها عبارتند از رنگها (با اجزای قرمز، سبز و آبی آنها بیان می‌شود) و مختصات (با اجزای x ، y بیان می شوند).

راه اطمینان از کوچک نگه داشتن رابط‌ها و شناسایی این توده داده‌ها با نگه داشتن تعریف تابع حداکثر تا چهار پارامتر است. روش استفاده از این دستورالعمل این است که دو یا چند آرگومان را در یک کلاس جدید قرار دهید و سپس از ارجاعات به این کلاس جدید استفاده کنید.

مثلا یک سیستم رزرو اتاق هتل و به عبارت دقیق‌تر، یک تابع برای دریافت قیمت برای اتاق‌های خاص در نظر بگیریم. از آنجا که در این دستورالعمل فقط با تعریف تابع‌ها سروکار داریم، متن اصلی تابع را نمی‌نویسیم:

public Quote getQuote(String hotelName, RoomType roomType,

                      boolean breakfastIncluded,

                      LocalDate checkInDate,

                      LocalDate checkOutDate)

این تابع پنج پارامتر دارد، یک پارامتر بیشتر از حد مجاز. ما می‌توانیم این مشکل را با بسته‌بندی تاریخ های ورود و خروج در کلاس TimePeriod برطرف کنیم.

public Quote getQuote(String hotelName, RoomType roomType,

                      boolean breakfastIncluded,

                      TimePeriod timePeriod) {

    // ... //

}

public class TimePeriod {

    public TimePeriod(LocalDate checkInDate,

     LocalDate checkOutDate) {

        // ... //

    }

}

نکته جالب در مورد کلاس TimePeriod این است که ما می‌توانیم به راحتی اعتبار سنجی را به آن اضافه کنیم: اطمینان حاصل کنید که تاریخ چک کردن حداقل یک روز پس از تاریخ اعلام حضور است، اطمینان حاصل کنید که تاریخ ورود در گذشته نیست و غیره. کوچک نگه داشتن رابط های واحد نه تنها تابع‌های ساده‌تر و قابل خواندن‌تری را ایجاد می‌کند، بلکه به ما در کپسوله کردن مفاهیم کمک می‌کند.

**مورد پنجم:**

منظور از ماژول‌ها یا کلاس‌ها، نمایش مفاهیم دامنه است. اگر نمی توانید کاری را که یک کلاس در چند جمله ساده انجام می‌دهد توضیح دهید، آن کلاس یا نشان دهنده بیش از یک مفهوم است یا مفهومی را نشان می‌دهد که بیش از حد کلی یا انتزاعی است.

کلاسی که مسئولیت بیش از حد داشته باشد از چند جهت دردسرساز خواهد بود. اول، این احتمال وجود دارد که به یک کانون تغییر تبدیل شود. از آنجا که مسئولیتهای زیادی دارد، بخش بزرگی از منطق تجارت را تحت تأثیر قرار خواهد داد و بنابراین احتمال نیاز به اصلاح آن در صورت درخواست جدید زیاد خواهد بود. تغییر نقاط اتصال مشکلات زیادی خواهد داشت.

دوم، کلاس‌های بزرگ خطر تبدیل شدن به محل تخلیه تصمیمات دشوار طراحی را دارند. وقتی لازم است قابلیت‌های جدیدی به سیستم اضافه شود و برنامه‌نویسان درباره اینکه این قابلیت جدید به کجا می‌رود مطمئن نیستند، غیر معمول نیست که افراد یک کلاس بزرگ موجود را انتخاب می‌کنند که هدف آنها به هر حال کاملاً مشخص نیست.

\*\*\*\*\*\*\*\* برای سایر موارد هم می‌توانیم مثالی بیاریم ولی تا همینجا هم تحلیل‌ها خارج از حد حوصله شده است اگر توضیحات کم است ببخشید. ☹

2) مقاله “[[2]](#footnote-2)”Architectural Mismatch: Why Reuse Is Still so Hard را مطالعه کنید.

الف) به نظر شما چالش­های مطرح شده در این مقاله بر طرف شده­اند یا همچنان معتبر هستند؟ توضیح دهید.

ب) چه پیشرفت­ها و تغییراتی در حوزه مهندسی نرم افزار (توسعه نرم افزار)، موجب برطرف شدن چالش های قبلی یا ایجاد چالش­های جدید شده است؟ توضیح دهید.

الف) موارد اشاره شده در این مقاله به شرح زیر میباشند:

* طبیعت خود مولفه ها
* طبیعت کانکتورها
* معماری جهانی
* فرآیند ساخت
* زیربنا و ساختاری که در آن مولفه ها ایجاد شده اند
* نرم افزار استفاده کننده از مولفه
* روابط بین مولفه ها

برخی از مشکلات با گذر زمان و بهبود تکنولوژی ها تا حدودی بهبود پیدا کرده اند اما برخی از مشکلات برخاسته از اصل موضوع استفاده مجدد است، این نوع از مشکلات همواره پابرجاست. مثلا یکی از راهکارهایی که به بهبود شرایط کمک میکند، استفاده از معماری های تخصصی است، اما همین که تصمیم بگیریم از یک مولفه در معماری تخصصی استفاده کنیم این نیاز را ایجاد میکند که مولفه را به آن معماری بقبولانیم، احتمالا سختی های ایجاد wrapper ها و... را در پی خواهد داشت. یکی دیگر از راهکارهایی که به بهبود شرایط کمک کرده است وجود open source community است این مسئله نیز به شرایط را برای استفاده مجدد بسیار بهبود داده است، اما چالشی که ایجاد کرده است این است که با ایجاد چنین شرایطی اعتماد به سیستم اوپن سورس کمی سخت است (هرچند همین متن باز بودن، سبب میشود بسیاری از حفره های امنیتی توسط community اعلام شود)

ب) چالش هایی که به همراه فناوری های جدید آمده است، یکی این است که با توجه به پیشرفت عجیب بستر اینترنت و منتقل شدن همه کارها حتی بانک داری روی این بستر، یکی از چالش های اصلی ای که ایجاد میشود اعتماد است، اگر بخواهیم همه جوانب امنیتی را فراهم کنیم، باید سربار آن را نیز بپذیریم و اگر بخواهیم امنیت را به خاطر سربارش کنار بگذاریم از اعتماد خود کاسته ایم، بنابراین بایستی برای استفاده از مولفه ها یک مصالحه ای انجام دهیم.

چالش دیگر این است که هر مولفه در طول عمر خود دچار تغییر میشود و این موضوع ناگزیر است بنابراین بایستی همواره با سیستم خودمان هماهنگ نگهش داریم، ضمنا بعد از هر آپدیت ممکن است اوایل دچار مشکل شود و در ریلیز های بعدی درست شود.

یکی از چالش های دیگری که با آن روبرو هستیم این است که این روزه اکثر مولفه ها امکان بازپیکربندی را به صورت پویا فراهم میکند که بتواند از تغییر نیازمندی و منابع مختلف و... پشتیبانی کند و این موضوع خود باعث میشود نیاز به مدیریت داشته باشد.

چالش آخر این است که این استفاده مجدد وقتی بخواهیم که مثلا معماری را تغییر دهیم ممکن است به صورت فلج کننده ای امکان تغییر معماری را از بین ببرد، در این حالت ها ممکن است هزینه سازگار کردن این مولفه به اندازه هزینه پیاده سازی آن باشد! بنابراین این چالش ها حتی با پیشرفت تکنولوژی هم به صورت جدی قابل حل شدن نیستند و علت آن این است که این مشکلات برخواسته از خود استفاده مجدد است و به ذات آن مربوط است.

3) دو دسته از ابزارهای مدیریت پیکربندی، ابزارهای مدیریت کد منبع و ابزارهای ساخت و ادغام پیوسته هستند.

الف) این دو دسته را مختصراً معرفی کنید.

ب) از هر دسته سه ابزار انتخاب کنید: ضمن معرفی هر ابزار، امکانات هریک را بررسی و با یکدیگر مقایسه کنید.

الف)

**مدیریت کد منبع:** اصطلاح مدیریت کد منبع[[3]](#footnote-3) یا کنترل نسخه[[4]](#footnote-4)، به عمل مدیریت، کنترل و ثبت و ضبط مستنداتی همچون کدها و مستندات نرم‎افزاری، اطلاعات وبسایت‌ها، تاریخچه‌ی تغییرات تصاویر و مواردی از این دست اطلاق می‌شود. تغییرات معمولا با یک شماره نسخه ذخیره می‌شوند. نرم‌افزارهای کنترل نسخه، پایه‌ی کارهای تیمی در تیم‌های نرم‌افزاری متشکل از چند توسعه‌دهنده است. در تیم‌های نرم‌افزاری، انتشار نسخه‌های مختلف از نرم‌افزار، توسعه بصورت تیمی و رفع کردن باگها بصورت اشتراکی و مواردی از این دست، همواره درحال انجام هستند. این امور بدون حضور نرم‌افزاری قدرتمند برای کنترل امور، بسیار سخت با پتانسیل بالای خطا و طاقت فرساست و البته نتیجه‌ی حاصل هم آنچه که باید باشد، نیست. سیستم‌های کنترل نسخه و مدیریت کد منبع امور ذکر شده را از هر زمانی آسانتر کرده‌اند.

**ساخت و ادغام پیوسته: C**ontinuous **I**ntegration یا به اختصار**CI** به طور خلاصه به پروسه‌ای اشاره دارد که از آن طریق فیچرهای جدید به صورت خودکار با مخزن اصلی ادغام می‌شوند اما**CD**  هم مخفف واژگان **C**ontinuous **D**elivery است و هم به **C**ontinuous **D**eployment اشاره دارد به طوری که اصطلاح اول به فرآیندی اشاره می‌کند که از آن طریق نرم‌افزار دائماً آماده دیپلوی است اما اصطلاح دوم سازوکاری است که به صورت خودکار کدهای آماده را روی سرور/سرورهای اصلی منتشر می‌کند.

[مدیریت کدمنبع — راهنمای گیت (codecast.ir)](http://git.codecast.ir/en/latest/book/01.whatis.html)

[CI/CD چیست؟ (sokanacademy.com)](https://sokanacademy.com/blog/10005/%D8%B3%DB%8C-%D8%A2%DB%8C-%D8%B3%DB%8C-%D8%AF%DB%8C-%DA%86%DB%8C%D8%B3%D8%AA)

ب)

مدیریت کدمنبع:

1- git: Git یکی از بهترین ابزارهای کنترل نسخه است که در بازار فعلی موجود است و تقریبا همه‎ی ما با آن کار کرده‎ایم. یک سیستم کنترل نسخه توزیع شده است که با هدف سرعت، یکپارچگی داده‌ها، پشتیبانی از گردش کار توزیع شده و غیرخطی انجام می‌شود. به غیر از مدیریت کد منبع نیز می‌توان از آن برای پیگیری تغییرات در هر مجموعه فایل استفاده کرد.

2- SVN: یک نسخه کنترل شده و منبع کنترل ابزار است که توسط آپاچی تولید شده است. این به توسعه دهندگان کمک می‎کند تا نسخه‎های مختلف کد منبع را حفظ کنند و یک تاریخچه کامل از همه تغییرات را حفظ کنند.

3- Mercurial: ابزاری برای کنترل کد است که به زبان پایتون نوشته شده است و برای توسعه‌دهندگان نرم‌افزار در نظر گرفته شده است. سیستم عامل‌هایی که پشتیبانی می‌کند unix-like، ویندوز و macOS است.

برای مقایسه این سه ابزار جدولی در پایین آمده است:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| نام ابزار | ویژگی‎ها | نقاط مثبت | نقاط منفی | متن‌باز |
| Git | پشتیبانی قوی برای توسعه غیر خطی فراهم می‌کند.  مدل مخزن توزیع شده  سازگار با سیستم‌ها و پروتکل‌های موجود مانندHTTP ، FTP ، ssh.  توانایی مدیریت کارآمد پروژه‌های کوچک تا بزرگ را دارد.  احراز هویت رمزنگاری تاریخ.  قابل اضافه کردن افزونه. | عملکرد فوق العاده سریع و کارآمد.  Cross platform  تغییرات کد را می‌توان به راحتی و به وضوح پیگیری کرد.  به راحتی قابل نگهداری و مقاوم است.  یک ابزار خط فرمان شگفت‌انگیز ارائه می‌دهد که به عنوان git bash شناخته می‌شود.  همچنین GUI رابط کاربری گرافیکی را ارائه می‌دهد | درک تاریخچه پیچیده و بزرگتر دشوار است.  از گسترش کلمه کلیدی و timestamp پشتیبانی نمی‌کند. | بله |
| SVN | مدل مخزن مشتری-سرور. با این حال، SVK به SVN اجازه می‌دهد تا شاخه‌های توزیع‌شده داشته باشد.  دایرکتوری‌ها نسخه‌بندی می‌شوند.  عملیات کپی، حذف، جابجایی و تغییر نام نیز نسخه شده است.  Metadata نسخه رایگان.  فضای ذخیره‌سازی باینری متفاوت  انشعاب به اندازه پرونده بستگی ندارد و این یک عملیات ارزان است. | از مزایای استفاده از ابزارهای رابط گرافیکی خوب مانند TortoiseSVN برخوردار است.  از پوشه‌های خالی پشتیبانی می‌کند.  در مقایسه با Git پشتیبانی از ویندوز بهتری دارد.  راه‌اندازی و مدیریت آن آسان است.  به خوبی با ویندوز، ابزارهای پیشرفته IDE و Agile ادغام می‌شود. | زمان اصلاح پرونده‌ها را ذخیره نمی‌کند.  با نرمال‌سازی نام پرونده به خوبی کار نمی‌کند.  از ویرایش‌های امضا شده پشتیبانی نمی‌کند. | بله |
| Mercurial | عملکرد بالا و مقیاس‌پذیری  قابلیت‌های پیشرفته انشعاب و ادغام.  توسعه مشترک توزیع شده  غیر متمرکز  هم متن ساده و هم پرونده‌های باینری را به صورت قوی مدیریت می کند.  دارای یک رابط وب یکپارچه است. | سریع و قدرتمند  آسان برای یادگیری  سبک و قابل حمل.  از نظر مفهومی ساده است | تمام موارد الحاقی باید در پایتون نوشته شوند.  وقتی با extensionهای اضافی استفاده می‎شود کاملاً مشکل ساز است. | بله |

[Top 5 BEST Version Control Software (Source Code Management Tools) (softwaretestinghelp.com)](https://www.softwaretestinghelp.com/version-control-software/)

ابزارهای CI/CD

1- Jenkins: یک ابزار crossplatform جاوا، منبع باز CI/CD است. این یکپارچه‌سازی مداوم، همراه با تسهیل تحویل مداوم را ارائه می‌دهد. همچنین تست و گزارش‌دهی بلادرنگ[[5]](#footnote-5) را امکان پذیر می‌کند. فقط با بارگیری موارد اجرایی آن در قالب .war و شروع همان کار از ترمینال می‌توان Jenkins را نصب کرد.

Jenkins Pipeline مجموعه‌ای از ابزارها را فراهم می‌کند که می تواند برای مدل‌سازی خطوط لوله انتقال به عنوان کد استفاده شود. Jenkins پایپ‌لاین را با استفاده از DSL (زبان خاص دامنه) پیاده‌سازی می‌کند. این یکی از پرکاربردترین و بهترین ابزارهای CI/CD است زیرا متن‌باز است و از مدت‌ها قبل وجود داشته است.

**ویژگی های برجسته جنکینز**

* برای سیستم‌عامل‌های ویندوز، لینوکس و macOS موجود است.

- رایگان و متن‌باز، درنتیجه برای استارتاپ‌ها و همچنین سازمان‌های مقیاس بزرگ‌ترجیح داده می‌شود.

- 16.3هزار star در گیت‌هاب و 6.5k فورک.

- قابل انعطاف

- با بیش از 1500 پلاگین و یک community عالی.

- با سیستم‌عامل‌های رایج ابر مانند AWS ، Google Cloud ، Azure ، Digital Ocean و موارد دیگر ادغام می‌شود.

- می‌توان برای انجام کار به صورت موازی و تحقق بخشیدن به نیازهای پیچیده CD استفاده کرد.

2- TeamCity: یک ابزار خط لوله CI / CD مبتنی بر سرور در جاوا است. توسط JetBrains ، شرکتی که پشت توسعه چندین ابزار مفید مانند PyCharm ، IntelliJ Idea و غیره است، توسعه و نگهداری می‌شود. این برنامه برای نصب بر روی سرورهای ویندوز و لینوکس موجود است.

TeamCity برای پروژه‌های متن‌باز رایگان است و به تیم‌های کوچک گزینه ساده‌ای را برای ادغام با Azure DevOps و Jira Software Cloud ارائه می‌دهد. این برنامه همچنین از راه‌اندازی عوامل ساخت در Kubernetes پشتیبانی می‌کند.

**ویژگی های برجسته TeamCity**

* قابل توسعه و قابل استفاده برای استفاده مجدد از تنظیمات یک پروژه در زیر پروژه.
* قابل اجرا سازه‌[[6]](#footnote-6)های موازی، در نتیجه انعطاف‌پذیری لازم را برای اجرای سازه‌ها به طور همزمان در سازه‌ها و محیط های مختلف فراهم می‌کند.
* خطوط لوله در TeamCity با استفاده از DSL مبتنی بر Kotlin (زبان خاص دامنه) تعریف می‌شوند.
* با Docker ، Visual Studio Team Services ، Maven ، NuGet و موارد دیگر ادغام می‌شود.
* نوع موجود آن با سیستم عامل های ابری معروف مانند Google Cloud ، AWS ، VMWare vSphere و موارد دیگر ادغام می‌شود.

3- CircleCI: یکی از بهترین ابزارهای CI/CD است که برای تحقق CI/CD برای پروژه‌های منبع‌باز و پروژه‌های بزرگ مناسب است. CircleCI Cloud ارائه شده مبتنی بر ابر است در حالی که سرور CircleCI پیش فرض self-host است.

از زبان‌هایی که می‌توانند روی سیستم عامل‌های ویندوز، لینوکس و macOS اجرا شوند پشتیبانی می‌کند. تنظیم آن آسان است و از YAML برای خطوط لوله آن استفاده می‌شود. در سال 2019 ، Forrester Wave از CircleCI به عنوان رهبر ادغام مداوم Cloud Native نام برد.

**ویژگی های برجسته CircleCI**

* راه اندازی آن آسان است و می‌تواند با سیستم های معروف کنترل‌کننده نسخه مانند GitHub ، Bitbucket و غیره استفاده شود.
* خطوط لوله CI/CD را به عنوان گزارش ارائه می‌دهد.
* اکثر زبان‌های برنامه‌نویسی محبوب را پشتیبانی می‌کند.
* برای کاهش زمان ساخت، سازه‌ها را می‌توان در چندین قسمت تقسیم و متعادل کرد.
* اجرای تست‌ها به صورت موازی در CircleCI.
* سرور CircleCI که پیشنهاد اولیه CircleCI است، می‌تواند با ابزارهای شخص ثالث معروف مانند GitHub Enterprise ، LambdaTest ، Coveralls و موارد دیگر ادغام شود.
* سرور CircleCI از سیستم‌عامل‌های ابری پرکاربرد مانند AWS ، Google Cloud ، Azure و موارد دیگر پشتیبانی می‌کند.

[27 Of The Best CI/CD Tools Available Today (lambdatest.com)](https://www.lambdatest.com/blog/27-best-ci-cd-tools/)

4) فعالیت­های پیکربندی نرم افزار را در حوزه ساخت برنامه های ‘نرم افزار به عنوان خدمت’[[7]](#footnote-7) بررسی کنید. مهمترین چالش­ها و مزایای آن را توضیح دهید.

پللتفرم‌های توسعه مبتنی بر ابر می‌توانند ترکیبی از پیکربندی و استقرارهای[[8]](#footnote-8) سفارشی را داشته باشند که به راحتی می‌تواند توسعه‌دهندگان را گیج کرده و استقرار[[9]](#footnote-9) را دشوار کند. برای حل این مسئله ، باید فرایند مدیریت پیکربندی خود را برای پروژه‌های مبتنی بر ابر سازگار کنید. در حالی که پیشنهادات SaaS به طور خاص برای محدود کردن زمان صرف شده در توسعه طراحی شده‌اند، برای ادغام کامل آنها با سیستم‌های معمولی سازمانی، به برخی از تغییرات اساسی نیاز است.

هنگام استفاده از بهترین روش‌ها برای مدیریت پیکربندی در محیط‌های مبتنی بر ابر، یک چالش بزرگ است زیرا:

* ابزارهای استقرار SaaS هنوز بالغ نشده‌اند
* تامین منابع ابری و مقیاس‎گذاری خودکار پیچیدگی را افزایش می‎دهد
* سرپرستان SaaS ممکن است با بهترین روش‌های مدیریت پیکربندی و انتشار آشنا نباشند
* ادغام SaaS اغلب به همگام‌سازی نسخه‌‌ها با سیستم‌های قدیمی نیاز دارد
* اسکریپت‌های پیکربندی، کد، لیست‌های دستی و استقرار باید در مخزن کد منبع مربوطه بررسی شود
* استقرار یک برنامه اغلب به مراحل دستی و خودکار نیاز دارد
* مشکلات موجود برای امنیت در فضای ابر
* مشکلات موجود برای در دسترس بودن در همه زمان‎ها با توجه به این مساله که این سرویس‎ها بر بستر ابر است
* و ...

علاوه بر چالش‎های ذکر شده، مزایایی نیز موجود است، از جمله:

* بهبود سریعتر خدمات با کمک تشخیص سریع
* به روزرسانی پویا و مداوم دارایی‎ها
* کنترل بهتر اکوسیستم IT با استفاده از ردیابی assetهای پویا[[10]](#footnote-10)
* بهبود عملکرد asset با تهیه و تهیه مجوز خودکار
* مدیریت کارآمد ظرفیت بیش از حد و کم
* بهینه سازی هزینه و منابع
* مدیریت بهتر برای بازیابی فاجعه
* مدیریت تغییرات موثر با چابکی[[11]](#footnote-11)

[Important SaaS Configuration Management Goals (walkme.com)](https://saasaddict.walkme.com/important-saas-configuration-management-goals/#:~:text=Principles%20of%20SaaS%20Configuration%20Management,available%20where%20and%20when%20needed.)

[Config-management (freshservice.com)](https://freshservice.com/assets/resources/freshservice/ebooks/Config-management.pdf)

* پاسخ تمرین ها را به زبان فارسی و به صورت تایپ شده، در قالب یک فایل Pdf، در مودل بارگزاری کنید.
* لطفا نظم، ساختار و توالی سوالات را در پاسخ ها رعایت کنید.
* **در هر پاسخ، منابع استفاده شده را درج نمائید.**
* فایل پاسخ تمرین را تنها با قالب **SE2-HW5-GroupNumber.pdf** در مودل بارگزاری کنید.
* بارگزاری تمرین توسط یکی از اعضاء گروه کافی است.
* برای پاسخ­های هر قسمت منابع استفاده شده را درج نمائید.
* فایل زیپ ارسال **نکنید**.
* به ازای هر روز تاخیر در تحویل تمرین 20% از نمره تمرین کسر خواهد شد.
* حداقل برخورد به پاسخ­های مشابه، تخصیص نمره کامل منفی به طرفین خواهد بود.

1. <https://cutt.ly/Jh8yxTA> [↑](#footnote-ref-1)
2. برای دریافت مقاله به قسمت “مطالعه بیشتر” در مودل مراجعه کنید. [↑](#footnote-ref-2)
3. Source Code Management [↑](#footnote-ref-3)
4. Version Control [↑](#footnote-ref-4)
5. Real time [↑](#footnote-ref-5)
6. build [↑](#footnote-ref-6)
7. Software as a Service (SaaS) [↑](#footnote-ref-7)
8. Deployment [↑](#footnote-ref-8)
9. Deploy [↑](#footnote-ref-9)
10. Dynamic asset tracking [↑](#footnote-ref-10)
11. Agile [↑](#footnote-ref-11)