پیشنویس: لطفاً پخش نکنید.

۱ مقدمه‌ای بر نرم‌افزار به‌صورت یک سرویس و توسعه چابک

۱-۱ مقدمه

این بخش هنوز نوشته نشده‌است و در نسخه‌های بعدی از این کتاب گنجانده خواهد شد. شما می توانید برای کسب اطلاع از بروزرسانی های آتی به وب سایت کتاب[1] مراجعه کنید.

۱-۲ فرایندهای تولید نرم‌افزار: طرح و سند

این بخش هنوز نوشته نشده‌است و در نسخه‌های بعدی از این کتاب گنجانده خواهد شد. شما می توانید برای کسب اطلاع از بروزرسانی های آتی به وب سایت کتاب[2] مراجعه کنید.

۱-۳ فرایندهای تولید نرم‌افزار: بیانیه روش چابک

این بخش هنوز نوشته نشده‌است و در نسخه‌های بعدی از این کتاب گنجانده خواهد شد. شما می توانید برای کسب اطلاع از بروزرسانی های آتی به وب سایت کتاب[3] مراجعه کنید.

۱-۴ معماری سرویس گرا

این بخش هنوز نوشته نشده‌است و در نسخه‌های بعدی از این کتاب گنجانده خواهد شد. شما می توانید برای کسب اطلاع از بروزرسانی های آتی به وب سایت کتاب[4] مراجعه کنید.

۱-۵ نرم‌افزار به عنوان یک سرویس

این بخش هنوز نوشته نشده‌است و در نسخه‌های بعدی از این کتاب گنجانده خواهد شد. شما می توانید برای کسب اطلاع از بروزرسانی های آتی به وب سایت کتاب[5] مراجعه کنید.

۱-۶ رایانش ابری

این بخش هنوز نوشته نشده‌است و در نسخه‌های بعدی از این کتاب گنجانده خواهد شد. شما می توانید برای کسب اطلاع از بروزرسانی های آتی به وب سایت کتاب[6] مراجعه کنید.

پیش‌نویس. لطفا پخش نکنید.

۱-۷ کد زیبا در میراث‌کد

این بخش هنوز نوشته نشده‌است و در نسخه‌های بعدی از این کتاب گنجانده خواهد شد. شما می توانید برای کسب اطلاع از بروزرسانی های آتی به وب سایت کتاب[7] مراجعه کنید.

۱-۸ تضمین کیفیت نرم‌افزار: تِست

و کاربران با خنده و طعنه گفتند:
«این دقیقاً همان چیزیست که ما درخواست کرده بودیم، اما نه آن چیزی که ما می‌خواهیم.»

—ناشناس

ما این مبحث را با ارائه تعریفی از کیفیت شروع می‌کنیم. تعریفی متعارف از ***کیفیت*** برای هر محصولی «در خور استفاده بودن»آن است، که باید هم برای تولیدکننده و هم برای مصرف‌کننده ارزش تجاری داشته باشد (جورن و گرینا، ۱۹۹۸). در مورد نرم‌افزار، کیفیت به معنی ارضای نیازهای مشتری (راحتی در استفاده، دریافت جواب‌های صحیح، عدم خروج ناگهانی و غیره) *و در عین حال سادگی* در بهبود نرم‌افزار و اشکال‌زدایی آن برای توسعه‌دهندگان است. ریشه ***تضمین کیفیت*** (***QA***) نیز به صنعت تولید برمی‌گردد، و به فرآیندها و معیارهایی که منجر به تولید محصولات با کیفیت بالا می‌شوند، و همچنین به ابداع فرآیندهای تولیدی که کیفیت را بالا می‌برند، اشاره دارد. پس تضمین کیفیت نرم‌افزار، به معنی حصول اطمینان از کیفیت بالای محصول درحال تولید و همچنین ابداع فرآیندها و استانداردهایی در یک سازمان است که به تولید محصولات با کیفیت بالا منجر می‌شود. همانطور که در ادامه خواهیم دید، در برخی از فرآیندهای نرم‌افزاریِ طرح-و-ثبت حتی از یک تیم جداگانه QA برای بررسی کیفیت نرم‌افزار استفاده می‌کنند (بخش ۱-۸).

تعیین کیفیت نرم‌افزار دو اصطلاح را شامل می‌شود که معمولاً بجای یکدیگر به کار می‌روند اما تفاوت ظریفی دارند (بوهم ۱۹۷۹):

- ***درستی‌سنجی***: آیا محصول را *درست* ساخته‌اید؟ (آیا محصول، طبق مشخصات از قبل تعریف شده ساخته شده؟)
- ***اعتبارسنجی***: آیا *محصولِ* درست را ساخته‌اید؟ (آیا این همان چیزی است که مشتری می‌خواهد؟ یا به عبارت دیگر، آیا مشخصاتِ با خواسته مشتری تطابق دارد؟)

پیش‌نمونه‌های نرم‌افزار که جوهره روش چابک هستند، عموماً بیشتر به اعتبارسنجی کمک می‌کنند تا به درستی‌سنجی، چونکه بسیاری از مشتریان با دیدن کارکرد محصول در مورد چیزی که می‌خواهند تغییر عقیده می‌دهند.

طریقه اصلی اعتبارسنجی و درستی‌سنجی، ***تِست*** است. انگیزه اصلی انجام تست یافتن هرچه زودتر اشتباهات توسط توسعه‌دهندگان است تا هزینه اصلاح کردن آن‌ها کاهش یابد. با توجه به فراوانی ترکیب‌های مختلف ورودی، انجام تست به‌صورت جامع ممکن نیست. یک راه برای کاهش فضای ممکن ورودی، انجام آزمون‌های مختلف در فازهای مختلف تولید نرم‌افزار است. به‌ترتیب از پایین به بالا، ابتدا ***تست واحد*** قرار می‌گیرد. این نوع تست به ما این اطمینان را می‌دهد که یک رویه، کاری را انجام می‌دهد که از آن انتظار می‌رود. سطح بعد ***تست ماژول*** است که تک‌تک واحدها را به‌صورت سراسری تست می‌کند. بطور مثال، تست واحد در محدوده یک کلاس کار می‌کند اما تست ماژول سرتاسر کلاس‌ها را بررسی می‌کند. بر روی این سطح، ***تست یکپارچگی*** قرار دارد، که تضمین می‌کند که رابطه‌های بین واحدها دارای فرضیات سازگاری هستند و به درستی باهم ارتباط برقرار می‌کنند. این سطح، به بررسی عملکرد یک واحد نمی‌پردازد. در بالاترین سطح، ***تست سیستم*** یا ***تست پذیرش*** قرار دارد که وظیفه‌ی بررسی برنامه کامل (که از ادغام واحدهای مختلف تشکیل شده است) را از جهت دارا بودن مشخصات تعیین شده بر عهده دارد. در فصل ۸، جایگزینی را برای تست معرفی می‌کنیم که ***روش‌های صوری*** نام دارد.

غیر عملی بودن تست جامع
فرض کنید که یک نانو ثانیه طول می‌کشد تا یک برنامه تست شود. این برنامه فقط یک ورودی ۶۴بیتی دارد که ما می‌خواهیم آن را بطور کامل تست کنیم. (بدیهی است که اکثر برنامه‌ها زمان بیشتری برای اجرا نیاز دارند و ورودی‌های آن‌ها نیز بیشتر است.) فقط همین مورد ساده 2^{64} نانو ثانیه یا ۵۰۰ سال طول خواهد کشید.

همانطور که به‌طور خلاصه در بخش ۳-۱ ذکر شد، تست در نسخه‌ی برنامه‌سازی اکستریم از روش چابک با نوشتن تست‌ها *قبل از* نوشتن کد انجام می‌پذیرد. پس از این کار باید با نوشتن کم‌ترین کد، تست را با موفقیت پشت سر گذاشت، این کار تضمین کننده‌ی کدی است که همواره تست می‌شود و احتمال تولید کدهای بی‌مصرف را کاهش می‌دهد. روش برنامه‌سازی اکستریم، فلسفه انجام تست در ابتدا را بسته به سطح تست به دو قسمت تقسیم می‌کند. برای تست سیستم، پذیرش و یکپارچگی، از ***طراحی رفتار محور*** (BDD) استفاده می‌کند که موضوع فصل ۷ است. و برای تست واحد و ماژول از ***توسعه آزمون محور*** (TDD) استفاده می‌کند که موضوع بحث فصل ۸ است.

خلاصه: انجام تست مخاطرات ناشی از خطا در طراحی را کاهش می‌دهد.

- به اشکال مختلف، تست کردن به ***سنجش درستی*** یک نرم‌افزار در داشتن مشخصات و همچنین ***سنجش اعتبار*** طراحی آن مطابق با خواسته‌های مشتری کمک می‌کند.
- با تقسیم تست به ***تست واحد***، ***تست ماژول***، ***تست یکپارچگی*** و ***تست سیستم*** یا ***پذیرش*** و تمرکز بر روی این قسمت‌ها در جهت غلبه بر غیرِعملی بودن یک تست جامع تلاش می‌شود. در این بخش‌ها هر کدام از سطوح بالاتر، انجام تست‌های جزئی را برعهده‌ی لایه‌ی زیرین می‌گذارند.
- برنامه‌سازی چابک نیز با استفاده از نوشتن تست‌ها پیش از نوشتن کد با استفاده از ***طراحی رفتار محور*** و ***توسعه آزمون محور*** سعی می‌کند بر سختی‌های تست غلبه کند.

خودآزمایی ۱-۸'۱. *می‌دانیم که تمامی روش‌های تست زیر در درستی سنجی به ما کمک می‌کنند؛ اما کدام مورد بیش از دیگران در اعتبار سنجی مفید است: واحد، ماژول، یکپارچگی، یا پذیرش؟*

◇ اعتبار سنجی به آنچه مشتری واقعا می‌خواهد می‌پردازد تا اینکه در مورد درست بودن کد و مشخصات آن به بحث بپردازد؛ بنابراین تست پذیرش بیش از همه‌ی موارد می‌تواند تفاوت بین کار را درست انجام دادن و کارِ درست را انجام دادن را تعیین کند. ■

■ ***بیشتر بدانیم: تست کردن: طرح-و-ثبت در مقابل چرخه‌های روش چابک***

در فرآیند توسعه آبشاری، پس از پایان هر فاز و در انتها در فاز درستی سنجی که شامل تست پذیرش نیز هست، تست انجام می‌گیرد. در مدل حلزونی، این اتفاق پس از هر چرخه می‌افتد که می‌تواند به یک تا دو سال به‌طول بیانجامد. تضمین در نسخه‌ی برنامه‌سازی اکستریم از روش چابک، تضمین با کمک BDD انجام می‌گیرد که در آن اگر قرار باشد از صفر کد تولید شود، تست‌ها *قبل* از تولید کد نوشته می‌شوند؛ و اگر بنا بر ارتقای کد دیگری باشد، نوشتن تمامی تست‌ها باید پیش از ارتقا دادن کد صورت پذیرد. میزان تست کردن به این بستگی دارد که آیا شما در حال ارتقای کدی تمیز هستید و یا کدی را در دست دارید که یک میراث‌کد است. مسلماً یک میراث‌کد به تست بیشتری نیاز دارد.

پس از این بررسیِ تضمین کیفیت، ببینیم که چگونه می‌توانیم توسعه‌دهندگانی بهره‌ورتر داشته باشیم.

۹-۱ بهره‌وری: اختصار، سنتز، بازکاربرد و ابزارها

امروزه بیشتر نرم‌افزارها مانند اهرام مصر هستند؛ مجموعه‌ای از خشت‌ها که صرفاً توسط هزاران برده و بی هیچ هوشمندی، بدون ساختاری یکپارچه بر روی هم جمع شده‌اند.

— آلن کِی، مجله *ACM Queue*، ۲۰۰۵

پیش‌نویس. لطفاً پخش نکنید.

بر اساس قانون مور منابع سخت‌افزاری نزدیک به ۵۰ سال است که در هر ۱۸ ماه قدرتی دو برابر پیدا می‌کنند. این رایانه‌های سریع با حافظه‌هایی بسیار بزرگتر قادر به اجرای نرم‌افزارهای بسیار بزرگتری نیز هستند. اما برای تولید نرم‌افزارهای بزرگی که بتوانند از ویژگی‌ها و قدرت این سیستم‌های قوی استفاده کنند، مهندسین نرم‌افزار نیاز داشتند تا عملکرد سودمندتری داشته باشند و بهره‌وری خود را بالا ببرند.
برای بالا رفتن بهره‌وری، مهندسان چهار سازوکار بنیادی را ایجاد کردند:

۱. وضوح به‌کمک اختصار

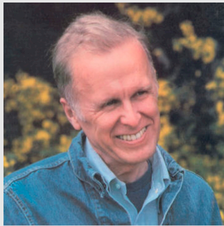
۲. سنتز

۳. بازکاربرد

۴. خودکارسازی به‌کمک ابزارها

یکی از اصلی‌ترین پیش‌فرض‌هایی که به افزایش بهره‌وری برنامه‌نویسان کمک می‌کند، خوانایی کد است. هر چه کد راحت‌تر فهمیده شود، اشکلات کمتری داشته و آسان‌تر تکامل پیدا می‌کند. مفهوم دیگری که در راستای همین اصل قرار می‌گیرد حجم برنامه است. هرچه برنامه کوچک‌تر باشد، فهمش آسان‌تر است. ما این مفهوم را با شعار «وضوح به‌کمک اختصار» بیان می‌کنیم.

جان بکوس (۱۹۲۴–۲۰۰۷) برای «مشارکت‌های بسیار زیاد، تاثیر گذار و ماندگارش در طراحی سیستم‌های کاربردیِ برنامه‌نویسیِ سطح بالا، به‌خصوص برای کارش برروی فرترن»، جایزه تورینگ را در سال ۱۹۷۷ دریافت کرد. زبان فورترن اولین زبان سطح بالایی است که بطور گسترده مورد استفاده قرار گرفت.



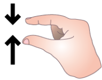
زبان‌های برنامه‌نویسی برای رسیدن به این هدف، دو روش را در پیش گرفتند. در روش اول، دستورات و ساختارهایی را در اختیار برنامه‌نویس قرار می‌دهند تا بتواند ایده‌هایش را به‌صورت طبیعی و با حروف کمتری بنویسد. برای مثال هر دو خط زیر یک مفهوم بایستگی ساده را نشان می‌دهند:

```
assert_greater_than_or_equal_to(a, 7)
a.should be >= 7
```

بدون شک خط دوم (که اتفاقاً به زبان روبی است) کوتاه‌تر، خواناتر و قابل فهم‌تر است، که این ویژگی‌ها نگه‌داری و توسعه‌ی آن را نیز بهبود می‌بخشند. در خط اول نه‌تنها ممکن است برای لحظه‌ای اولویت آرگومان‌ها باعث سردرگمی شود، بلکه خواندن عبارتی طولانی‌تر که تعداد حروفش دو برابر است، بار ادراکی مضاعفی را برای خواننده ایجاد می‌کند (به فصل ۳ رجوع کنید).

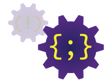
روش دیگر برای افزایش وضوح، بالا بردن سطح انتزاع است. این روش در ابتدا به تولید زبان‌های سطح بالاتری مانند فورترن و کوبال منجر شد. ظهور این زبان‌ها باعث ارتقای سطح مهندسی نرم‌افزار از زبان اسمبلی برای یک رایانه خاص، به زبان‌های سطح بالایی شد، که می‌توانستند تنها با عوض کردن کامپایلر برروی رایانه‌های مختلف اجرا شوند.

با ادامه روند رو به رشد عملکرد سخت‌افزار، تعداد بیشتری از برنامه نویسان مشتاق بودند تا کارهایی را که قبلا خودشان انجام می‌دادند به کامپایلر و سیستم‌های زمان اجرا واگذار کنند. به‌طور مثال جاوا و زبان‌های مشابه، مدیریت حافظه را به‌دست گرفتند. ویژگی‌ای که در زبان‌های قدیمی‌تری مانند C و C++ بر عهده‌ی برنامه‌نویس گذاشته شده‌بود. زبان‌های اسکریپتی مانند پایتون و روبی سطح انتزاع را از این هم بالاتر برده‌اند. مواردی مانند ***بازتاب***، که به برنامه اجازه می‌دهد تا خودش را تحت نظر بگیرد و یا ***فرابرنامه‌نویسی***، که به برنامه قابلیت ایجاد تغییر در ساختار و رفتار خود را در زمان اجرا می‌دهد، نمونه‌هایی از این انتزاع سطح بالاتر هستند. ما برای نشان دادن مثال‌هایی که بهره‌وری را به‌کمک اختصار بالا می‌برند از این نماد «مختصر» استفاده می‌کنیم.



دومین راهکار مربوط به بهره‌وری، سنتز است؛ به این معنی که پیاده‌سازی به جای اینکه به صورت دستی انجام بگیرد به صورت خودکار تولید می‌شود. سنتزِ مدار منطقی برای مهندسین سخت‌افزار به این معناست که آن‌ها بتوانند یک سخت‌افزار را به‌صورت توابع بولیِ توصیف کنند و در ازای آن ترانزیستورهایی بهینه شده که توابع مورد نظر را پیاده کرده‌اند، تحویل بگیرند. مثال کلاسیک در سنتز نرم‌افزاری عملیات ***Bit blit*** است. این دستور گرافیکی وظیفه‌ی ترکیب دو بیت‌مپ مختلف تحت کنترل یک ماسک را برعهده دارد. روش ابتدایی حل این مسأله، یک عبارت شرطی برای انتخاب ماسک در داخلی‌ترین حلقه داشت، که البته روش کندی بود. راه حل، نوشتن برنامه‌ای بود که می‌توانست کد خاص را *بدون* داشتن عبارت شرطی داخل حلقه سنتز کند. ما برای مشخص

پیش‌نویس: لطفاً پخش نکنید.



کردن مثال‌هایی که با تولید خودکار کد باعث افزایش بهره‌وری می‌شوند، از این نماد چرخدنده‌های «تولید کد» استفاده می‌کنیم.

سومین راهکار بهره‌وری این است که به‌جای نوشتن همه‌چیز از ابتدا، از قسمت‌هایی که پیش‌تر طراحی شده‌اند، مجدداً استفاده کرد. از آنجایی که ایجاد تغییرات کوچک در نرم‌افزار آسان‌تر از سخت‌افزار است، امکان بازکاربری از قسمت‌هایی که حتی نتوانند نیازهای نرم‌افزار را به‌طور کامل برطرف کنند، بیشتر از سخت‌افزار است. ما مثال‌هایی که برای افزایش بهره‌وری از بازکاربرد استفاده می‌کنند را با این نماد «بازکاربرد» که شبیه نماد بازیافت است، مشخص می‌کنیم.

رَویه‌ها و توابع برنامه نویسی در اولین روزهای نرم‌افزار ابداع شدند تا بخش‌های مختلف یک برنامه بتوانند از یک کد یکسان با پارامترهای متفاوت استفاده کنند. کتابخانه‌های استانداردی برای ورودی/خروجی و توابع ریاضی به دنبال آن به‌وجود آمدند تا برنامه‌نویسان بتوانند از کدهای دیگران نیز استفاده (مجدد) کنند. رَویه‌هایی که در کتابخانه‌ها وجود دارند، امکان استفاده مجدد از پیاده‌سازی یک کار مشخص را به‌وجود می‌آورند. ولی بیشتر برنامه‌نویسان می‌خواهند از ***مجموعه‌هایی*** از کارها بازکاربری کرده و آن‌ها را مدیریت کنند. قدم بعدی در بازکاربری نرم‌افزاری، ***برنامه‌نویسی شیءگرا*** بود، که در آن می‌توان یک کار را (به‌کمک ارث‌بری در زبان‌هایی مانند ++C و جاوا)، در اشیا مختلف مجدداً مورد استفاده قرار داد.

اگرچه ارث‌بری امکان استفاده مجدد از پیاده‌سازی‌ها را فراهم می‌کرد، وجود یک روش کلی برای انجام کارهایی که حتی پیاده‌سازی‌شان متفاوت بود نیز فرصتی مناسب برای بازکاربرد ایجاد می‌کرد. این‌جا بود که ***الگوهای طراحی*** که از معماری ساختمانی الهام گرفته شده بودند (الکساندر و همکاران، ۱۹۹۷)، سربرآوردند تا پاسخگوی این نیاز باشند. زبان‌های برنامه‌نویسی برای پشتیبانی از بازکاربردِ الگوهای طراحی امکاناتی را به خود افزودند. ***نوع‌دهی پویا*** یکی از این امکانات بود که ترکیب انتزاع‌ها را آسان می‌ساخت. ***mix-in ها*** دیگر امکانی بودند که استفاده از کارکرد چندین مِتُد را بدون نیاز به ارث‌بری چندگانه و برخی از نابه‌هنجاری‌های دیده‌شده آن، فراهم می‌ساختند. پایتون و روبی از نمونه زبان‌هایی هستند که امکاناتی را فراهم می‌کنند تا به بازکاربردِ الگوهای طراحی کمک کنند.

توجه داشته باشید که بازکاربرد به این معنی *نیست* که شما تکه‌کدی را کپی کنید و آن را عیناً در چندین محل قرار دهید. مشکل کپی کردن این است که در هنگام رفع یک اشکال و یا افزودن یک ویژگی جدید ممکن است همه‌ی نسخه‌های کپی شده را به‌روز نکنید. یک راهبرد مهندسی نرم‌افزار در مخالفت با تکرار این است:

> *«هر قسمتی از اطلاعات موجود در یک سیستم باید تنها دارای یک نماینده نامبهم و معتبر باشد.»*
>
> —آندی هانت و دیوید توماس، ۱۹۹۹

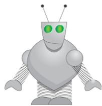
این رهنمود در این شعار خلاصه شده است: ***خودت را تکرار مکن!*** (***DRY***). از این پس ما برای مشخص کردن DRY از نماد حوله استفاده می‌کنیم.

یکی از کارکردهای اساسی مهندسی کامپیوتر یافتن راه‌هایی برای جایگزین کردن کارهای خسته‌کننده‌ی دستی با ابزارها است. این روند به صرفه‌جویی در زمان، افزایش دقت و یا هردوی این‌ها کمک می‌کند. واضح‌ترین ابزارهای طراحی به‌کمک کامپیوتر (CAD) برای توسعه نرم‌افزار، کامپایلرها و مفسرها هستند که سطح انتزاع را بالا می‌برند و همانطور که پیشتر اشاره شد، کد نیز تولید می‌کنند. اما ابزارهای بهره‌وری ماهرانه‌تری مانند Makefile ها و سیستم‌های کنترل نسخه (به بخش ۱-۱۰ مراجعه کنید) نیز وجود دارند که کارهای خسته‌کننده را به صورت خودکار انجام می‌دهند. ما مثال‌های مربوط به ابزارها را با نماد چکش مشخص می‌کنیم.

یکی از نکات مهم، زمان مورد نیاز برای یاد گرفتن طریقه استفاده از یک ابزار در مقابل میزان صرفه‌جویی در وقت در صورت استفاده از آن است. قابل اعتماد بودن ابزار، کیفیت تجربه‌ی کاربری و تصمیم‌گیری در انتخاب یک ابزار از میان چندین گزینه، از دیگر موارد حائز اهمیتی هستند که می‌توان به آن‌ها اشاره کرد. با این وجود، یکی از باورهای بنیادی مهندسی نرم‌افزار این است که یک ابزار جدید می‌تواند زندگیمان را بهتر کند.

نویسندگان این کتاب نیز به ارزش خودکارسازی و ابزارها را معتقدند. به‌همین دلیل در طول این کتاب شما را با ابزارهایی آشنا خواهیم کرد تا بهره‌وریتان را افزایش دهیم. خوشبختانه هر ابزاری که در این کتاب به شما

معرفی می‌شود، به‌دقت مورد بررسی قرار گرفته است تا قابل اطمینان باشد. همچنین زمانی را که صرف آموختن آن می‌کنید، در ادامه با صرفه‌جویی‌ای که در زمان توسعه ایجاد می‌شود جبران خواهد شد و افزایش کیفیت را نیز به همراه خواهد داشت. برای مثال در فصل ۷ نشان می‌دهیم که چطور ***Cucumber*** به‌صورت خودکار روایت‌های کاربر را به تست‌های یکپارچگی تبدیل می‌کند و همچنین اینکه چطور ***Pivotal Tracker*** به‌صورت خودکار ***سرعت*** که مقیاسی برای نرخ افزوده شدن قابلیت‌ها به نرم‌افزار است، را محاسبه می‌کند. در فصل ۸ به معرفی ***RSpec*** که خودکارسازی فرآیند تست واحد را انجام می‌دهد، می‌پردازیم. متاسفانه شما باید کار با چندین ابزار جدید را یاد بگیرید، که البته ما فکر می‌کنیم توانایی فراگیری و استفاده‌ی سریع از ابزارها یک نیاز برای موفقیت در مهندسی نرم‌افزار است؛ بنابراین این مهارت، زمینه خوبی برای سرمایه‌گذاری است.



پس چهارمین راهکار برای افزایش بهره‌وری، خودکارسازی کارها به‌کمک ابزارهاست. ما مثال‌هایی که از خودکارسازی استفاده می‌کنند را با نماد ربات نمایش می‌دهیم؛ اگرچه بیشتر این مثال‌ها به ابزارها مرتبط‌اند.

خلاصه: قانون مور به مهندسین نرم‌افزار الهام بخشید تا بهره‌وریشان را به‌کمک موارد زیر افزایش دهند:

- میل به اختصار، با استفاده از قواعدی خلاصه‌تر و با بالا بردن سطح طراحی به‌کمک استفاده از زبان‌های سطح بالاتر. پیشرفت‌های اخیر در این زمینه شامل ***بازتاب*** که به برنامه اجازه می‌دهد تا خودش را تحت نظر بگیرد، و ***فرابرنامه‌نویسی*** که به برنامه قابلیت ایجاد تغییر در ساختار و رفتار خود را در زمان اجرا می‌دهد، است.
- سنتز پیاده‌سازی‌ها.
- بازکاربردِ طراحی‌ها با استفاده از اصل ***خودت را تکرار مکن! (DRY)*** و با تکیه بر نوآوری‌هایی مانند رَویه‌ها، کتابخانه‌ها، برنامه‌نویسی شیءگرا و الگوهای طراحی که به استفاده مجدد کمک می‌کنند.
- استفاده (و ایجاد) ابزارهای CAD برای خودکارسازی کارهای خسته‌کننده.

خودآزمایی ۱-۹-۱. *کدام راهکار ضعیف‌ترین دلیل برای مزیت‌های افزایش بهره‌وری در استفاده از کامپایلرها برای زبان‌های سطح بالا است؟ وضوح به‌کمک اختصار، سنتز، بازکاربرد یا استفاده از ابزارها برای انجام خودکار کارها؟*

◇ کامپایلرها استفاده از زبان‌های سطح بالا را ممکن می‌سازند و به برنامه‌نویسان اجازه می‌دهند تا بهره‌وری خود را با استفاده از ساختارهای کوتاه‌تر موجود در این زبان‌ها افزایش دهند. کامپایلرها کد سطح بالا را به‌عنوان ورودی گرفته، و کد سطح پایین معادل آن را تولید(سنتز) می‌کنند. کامپایلرها قطعاً ابزار نیز هستند. با اینکه شما می‌توانید ادعا کنید که بازکاربرد به کمک زبان‌های سطح بالا آسان‌تر است، اما بازکاربرد ضعیف‌ترین دلیل برای توضیح مزیت‌های استفاده از کامپایلرهاست. ■

■ ***بیشتر بدانیم: بهره‌وری: طرح-و-ثبت در مقابل چرخه‌های روش چابک***

بهره‌وری با مقیاس مهندس-ساعت برای پیاده‌سازی یک تابع جدید سنجیده می‌شود. تفاوت در این است که چرخه‌ها در روش آبشاری و حلزونی به نسبت روش چابک بسیار طولانی‌تر هستند (۶ تا ۲۴ ماه در مقابل نیمی از ماه). درنتیجه در فاصله بین ارائه نتایج به مشتری کارهای بیشتری انجام می‌شود و احتمال رد شدن قسمت بیشتری از کارها از سمت مشتری افزایش می‌یابد.

پیش‌نویس. لطفاً پخش نکنید.

۱-۱۰ گذری بر این کتاب

این بخش هنوز نوشته نشده‌است و در نسخه‌های بعدی از این کتاب گنجانده خواهد شد. شما می توانید برای کسب اطلاع از بروزرسانی های آتی به وب سایت کتاب[۸] مراجعه کنید.

۱-۱۱ چگونه این کتاب را نخوانیم

این بخش هنوز نوشته نشده‌است و در نسخه‌های بعدی از این کتاب گنجانده خواهد شد. شما می توانید برای کسب اطلاع از بروزرسانی های آتی به وب سایت کتاب[۹] مراجعه کنید.

۱-۱۲ اشتباهات و باورهای غلط

این بخش هنوز نوشته نشده‌است و در نسخه‌های بعدی از این کتاب گنجانده خواهد شد. شما می توانید برای کسب اطلاع از بروزرسانی های آتی به وب سایت کتاب[۱۰] مراجعه کنید.

۱-۱۳ نتیجه‌گیری: مهندسی نرم‌افزار بیش از برنامه‌نویسی است

این بخش هنوز نوشته نشده‌است و در نسخه‌های بعدی از این کتاب گنجانده خواهد شد. شما می توانید برای کسب اطلاع از بروزرسانی های آتی به وب سایت کتاب[۱۱] مراجعه کنید.

پیش‌نویس. لطفاً پخش نکنید.