واحد شهر مجلسی

مبانی مهندسی نرم افزار

فهرست مطالب

فصل ۱ : محصول ۴

فصل ۲ : فرآیند ۱۲

فصل ۳ : مفاهیم مدیریت پروژه ۳۴

فصل ۴ : معیارهای پروژه و فرآیند نرم افزار ۴۴

فصل ۱۳ : اصول و مفاهیم طراحی ۶۱

فصل ۱۷ : تکنیک های آزمون نرم افزار ۶۸

فهرست تصاویر

شکل ۱-۱ منحنی شکست سخت افزار ۶
شکل ۲-۱ منحنی های شکست واقعی و ایده آل برای نرم افزار ۶
شکل ۳-۱ تأثیر تغییر ۱۱
شکل ۱-۲ لایه های مهندسی نرم افزار ۱۳
شکل ۳-۲ .... ۱۹
الف) فازهای یک حلقه حل مسئله [RAC95]. ب) فازهای موجود در داخل فازهای حلقه حل مسئله [RAC95]. ۱۹
شکل ۴-۲ مدل ترتیبی خطی ۲۰
شکل ۵-۲ الگوی ساخت نمونه اولیه ۲۲
شکل ۶-۲ مدل RAD ۲۵
شکل ۷-۲. مدل گام به گام ۲۶
شکل ۸-۲ مدل مارپیچی معمولی ۲۸
شکل ۹-۲ مدل مارپیچی WINWIN ۳۰
شکل ۱۱-۲ بسط مبتنی بر مولفه ها ۳۱
شکل ۲-۳ تلفیق کردن مسئله و پروژه ۴۱
شکل ۱-۴ مواردی که کیفیت نرم افزار و کارآیی سازمانی را تعیین می کنند. ۴۷
شکل ۲-۴ علل نقایص و منشاء آن برای چهار پروژه نرم افزاری ۴۹
شکل ۳-۴ نمودار استخوان ماهی (برگرفته از [GRA92]). ۴۹
شکل۴-۴ معیارهای اندازه گرا ۵۲
شکل ۵-۴ محاسبه نقاط عملکرد ۵۳
شکل ۶-۴ محاسبه شاخص نقطه عملرد سه بعدی ۵۵
شکل ۱-۱۳ تبدیل تحلیل به طراحی نرم افزار ۶۲
شکل ۲-۱۳ مدولاریته و هزینه نرم افزار ۶۷
شکل ۱-۱۷ نشانه گذاری گراف گردشی ۷۹
شکل ۲-۱۷ الف نمودار گرشی ب گراف جریان ۷۹
شکل ۳-۱۷ منطق مرکب ۸۰
شکل ۴-۱۷ PDL برای طراحی موارد آزمون که در آن گره ها تعیین شده است ۸۳
شکل ۵-۱۷ گراف گردشی برای رویه average ۸۵
شکل ۶-۱۷ ماتریس گراف ۸۷
شکل ۷-۱۷ ماتریس ارتباط ۸۸
شکل ۸-۱۷ انواع حلقه ها ۹۴
شکل ۹-۱۷ الف نشانه گذاری گرافی ب یک مثال ساده ۹۸
شکل ۱۰-۱۷ یک نمای هندسی از موارد آزمون ۱۰۵
شکل ۱۱-۱۷ یک آرایه ارتوگونال L9 .... ۱۰۶

فصل ۱: محصول

نرم افزار چیست؟ نرم افزار کامپیوتری محصولی است که مهندس نرم افزار طراحی می کند و می‌سازد. شامل برنامه هایی است که در کامپیوتری به هر اندازه و با هر معماری قابل اجرا هستند، مستنداتی دارد که شامل فرم های کپی شده و مجازی می شود و داده هایی دارد که ترکیبی از ارقام و حروف است و البته می تواند شامل اشکال نمایشی از قبیل اطلاعات تصویری، ویدیویی و صوتی باشد.

چه می کند؟ مهندسین نرم افزار آن را می سازند و در حقیقت هر کسی در دنیای صنعت چه مستقیم و چه غیر مستقیم از آن استفاده می کند.

چرا اهمیت دارد؟ چون تقریباً همهٔ جنبه های زندگی ما را تحت تأثیر قرار می دهد و در تجارت، فرهنگ و فعالیتهای روزمره ما نمایان است.

چه مراحلی دارد؟ نرم افزارهای کامپیوتری نیز همانند تمام محصولات موفق دیگر ساخته می شوند، یعنی با اجرای فرآیندی که منجر به نتیجه‌ای با کیفیت بالا می شود و نیازهای کاربران آن را برآورده می سازد. شما روش مهندسی نرم افزار را به کار خواهید بست.

محصول کار چیست؟ از دیدگاه مهندس نرم افزار، محصول کار، برنامه ها، مستندات و داده ها هستند که نرم افزار کامپیوتری است. ولی از دیدگاه کاربر، محصول کار، اطلاعاتی است که به نحوی به درد کاربر می خورند.

چطور مطمئن شوم که درست از عهده کار برآمده‌ام؟ بقیهٔ کتاب را بخوانید، ایده های قابل اجرا روی نرم افزار خود را انتخاب کنید و آن را در کار خود اجرا نمایید.

طی پنج سالی که از نوشته شدن ویرایش چهارم این کتاب می گذرد، نقش نرم افزار به عنوان نیروی محرکه، نمود بیشتری یافته است. پدیده های نرم افزاری موسوم به اینترنت، اقتصاد ۵۰۰ میلیارد یورو را به خود اختصاص داده است. سرمایه داران وال استریت، سرمست از پیروزی یک الگوی اقتصادی جدید، پیش از آن که شرکتهای کوچک "دات کام" حتی یک دلار عایدی داشته باشند، ارزشهای میلیارد دلاری برای آنها تعیین کردند. صنایع متکی به نرم افزار جدیدی پدید آمده‌اند و آنها که قادر نیستند خود را با این نیروی محرکه جدید تطبیق دهند در خطر نابودی قرار دارند. دولت آمریکا علیه بزرگترین شرکت نرم افزاری اعلام جرم کرده است، درست همان طور که در دوران قبل نسبت به انحصارگرایی در صنایع نفت و فولاد عمل کرده بود.

تأثیر نرم افزار بر فرهنگ و جامعه ما همچنان در حال قوت گرفتن است. هرچه بر اهمیت آن افزوده شود، جامعهٔ نرم افزاری می کوشد فنّاوریهایی را توسعه بخشد که ساخت برنامه های کامپیوتری با کیفیت بالاتر را آسانتر، سریعتر و کم هزینه‌تر کند.

در این کتاب چارچوبی ارائه می شود که سازندگان نرم افزار می توانند از آن استفاده کنند. این فنّاوری شامل یک فرآیند، یک مجموعه روشها و آرایه‌ای از ابزارها می شود که آن را مهندسی نرم افزار می گویند.

۱-۱ نقش تکاملی نرم افزار

امروزه نرم افزار نقشی دوگانه دارد. نرم افزار نوعی محصول است و در عین حال وسیلهٔ نقلیه‌ای برای تحویل یک محصول، به عنوان محصول، توان محاسباتی بالقوهٔ یک سخت افزار کامپیوتری یا به طور گسترده‌تر، شبکه‌ای از کامپیوترها را بالفعل می کند. نرم افزار چه در داخل یک تلفن همراه باشد و چه درون یک کامپیوتر بزرگ عمل کند، یک مبدل اطلاعات است. تولید، مدیریت، اکتساب، اصلاح، نمایش یا انتقال اطلاعاتی که می تواند به سادگی یک بیت

باشد یا به پیچیدگی یک نمایش چند رسانه‌ای. نرم افزار به عنوان وسیله نقلیه‌ای برای تحویل یک محصول، مبنای کنترل کامپیوتر (سیستم عامل)، ارتباط اطلاعاتی (شبکه ها) و خلق و کنترل برنامه های دیگر (محیط ها و ابزارهای نرم افزاری) را تشکیل می دهد.

- چرا به پایان رساندن یک نرم افزار این قدر وقت می گیرد؟
- چرا ساخت نرم افزار هزینۀ بالایی دارد؟
- چرا نمی توانیم همۀ خطاها را پیش از تحویل نرم افزار به مشتری بیابیم؟
- چرا در اندازه گیری پیشرفت ساخت نرم افزار، دچار مشکل می شویم؟

۱-۲ نرم افزار

در سال ۱۹۷۰، کمتر از یک درصد عامۀ مردم می توانستند توصیفی هوشمندانه از مفهوم «نرم افزار کامپیوتری» ارائه دهند. امروزه اکثر حرفه‌ای ها و بسیاری از افراد عامه، سخت معتقدند که می دانند «نرم افزار» چیست. ولی آیا واقعاً می دانند؟

۱-۲-۱ ویژگیهای نرم افزار

برای درک مفهوم نرم افزار (و سرانجام درکی از مهندسی نرم افزار)، بررسی آن دسته از ویژگیهای نرم افزار که آن را از دیگر چیزهای ساخته دست بشر متمایز می سازد، اهمیت دارد. هنگامی که سخت افزاری ساخته می شود، فرآیند آفرینش بشری (تحلیل، طراحی، ساخت، آزمون)، سرانجام به یک شکل فیزیکی منتهی می شود. اگر یک کامپیوتر جدید می سازیم، طرحهای اولیه، ترسیمات طراحی رسمی و نمونه های اولیه به یک محصول فیزیکی (تراشه ها، مدارها، منبع تغذیه و غیره) تکامل می یابند.

نرم افزار یک عنصر سیستمی منطقی است نه فیزیکی. از این رو، نرم افزار دارای ویژگیهایی است که تفاوت چشمگیری با ویژگیهای سخت افزار دارند.

۱. نرم افزار، مهندسی و بسط داده می شود و چیزی نیست که به معنای کلاسیک کلمه، ساخته شود.

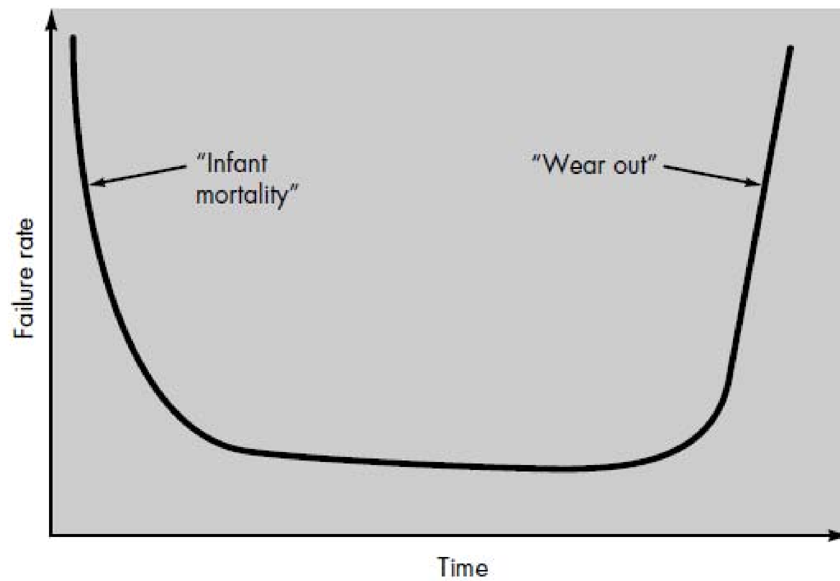
گرچه شباهتهایی میان بسط نرم افزار و ساخت سخت افزار وجود دارد، این دو عمل تفاوت بنیادی دارند. در هر دو عمل، کیفیت بالا از طریق طراحی خوب به دست می آید، ولی فاز ساخت برای سخت افزار باعث بروز مشکلات کیفیتی می شود که برای نرم افزار وجود ندارند (یا به راحتی قابل رفع هستند). هر دو عمل وابسته به انسان هستند، ولی رابطه میان انسان و کاری که انجام می شود، کاملاً متفاوت است. هر دو عمل مستلزم ساخت یک «محصول» هستند ولی روشها متفاوت است.

هزینه های نرم افزار در مهندسی آن متمرکز است. این بدان معناست که پروژه های نرم افزاری را نمی توان همانند پروژه های تولید معمولی مدیریت کرد.

۲. نرم افزار فرسوده نمی شود.

شکل ۱-۱ نمودار آهنگ شکست را به صورت تابعی از زمان برای سخت افزار نشان می دهد. این رابطه که غالباً «منحنی وانی» نامیده می شود، نشان می دهد که سخت افزار، آهنگ شکست نسبتاً شدیدی در ابتدای عمر خود نشان می دهد (این شکستها را غالباً می توان به عیوب طراحی و تولید نسبت داد)؛ این عیوب تصحیح می شوند و آهنگ شکست برای یک دورۀ زمانی به حدی ثابت نزول می کند (که امید می رود، بسیار پایین باشد). با گذشت زمان، سخت افزار شروع به فرسایش کرده دوباره آهنگ شکست شدت می گیرد.

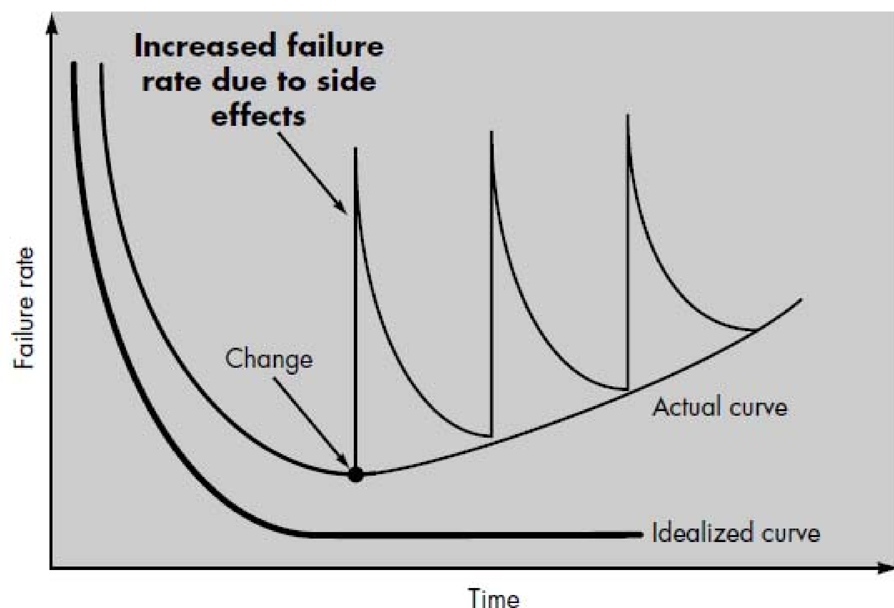
FIGURE 1.1
Failure curve for hardware



شکل ۱-۱ منحنی شکست سخت افزار

نرم افزار نسبت به ناملایمات محیطی که باعث فرسایش نرم افزار می شود، نفوذپذیر نیست. بنابراین، در تئوری، منحنی شکست برای نرم افزار باید شکل منحنی ایده آل شکل ۲-۱ را به خود بگیرد. عیوب کشف نشده باعث آهنگ شکست شدید، در ابتدای عمر برنامه می شود. ولی، این عیوب برطرف می شوند (با این امید که خطاهای دیگر وارد نشود) و منحنی به صورتی که نشان داده شده است، هموار می شود. منحنی ایده آل نسبت به منحنی واقعی مدلهای شکست نرم افزار، بسیار ساده‌تر است. ولی، معنای آن واضح است، نرم افزار هرگز دچار فرسایش نمی شود بلکه فاسد می شود!

FIGURE 1.2
Idealized and actual failure curves for software



KEY POINT
Software engineering methods strive to

شکل ۲-۱ منحنی های شکست واقعی و ایده آل برای نرم افزار

این تناقض ظاهری را می توان با در نظر گرفتن «منحنی واقعی» به بهترین وجه توضیح داد (شکل ۲-۱). نرم افزار در دوران حیات خود دستخوش تغییر می شود (نگهداری). با اعمال این تغییرات، احتمال دارد که برخی عیوب جدید وارد شوند و باعث خیز منحنی آهنگ شکست شوند (شکل ۲-۱). پیش از آنکه منحنی بتواند به آهنگ شکست منظم اولیه خود برسد، تغییر دیگری درخواست می شود که باعث خیز دوباره منحنی می شود. حداقل میزان شکست به آهستگی افزایش می یابد- نرم افزار در اثر تغییر فاسد می شود.

یک جنبهٔ دیگر از فرسایش نیز اختلاف میان سخت افزار و نرم افزار را نشان می دهد. هنگامی که یک قطعه از سخت افزار فرسوده می شود، با یک قطعه یدکی تعویض می شود. ولی نرم افزار قطعات یدکی ندارد. هر شکست نرم افزاری نشانگر خطایی در طراحی یا فرآیندی است که طراحی از طریق آن به کدهای قابل اجرا روی ماشین تبدیل می شود. از این رو، نگهداری نرم افزار به مراتب پیچیده‌تر از نگهداری سخت افزار است.

۳. گرچه صنعت در حال حرکت به سوی مونتاژ قطعات است، اکثر نرم افزارها همچنان به صورت سفارشی ساخته می شوند.

شیوه‌ای را در نظر بگیرید که در آن سخت افزار کنترلی برای یک محصول کامپیوتری طراحی و ساخته می شود. مهندس طراح یک الگوی ساده از مدار دیجیتالی رسم می کند، قدری تحلیل بنیادی انجام می دهد تا از عملکرد صحیح اطمینان حاصل کند، و سپس به قفسهٔ حاوی کاتالوگهای قطعات رجوع می کند. پس از انتخاب همهٔ قطعات می تواند آنها را سفارش دهد.

به موازات تکامل یک رشته مهندسی، مجموعه‌ای از قطعات طراحی استاندارد ایجاد می شود. پیچ های استاندارد و مدارات مجتمع فقط دو مورد از هزاران قطعهٔ استانداردی هستند که مهندسان مکانیک و برق در طراحی سیستمهای جدید به کار می برند. قطعات قابل استفاده مجدد طوری طراحی شده‌اند که مهندس بتواند بر عناصر واقعاً جدید یک طراحی، یعنی قطعاتی از طراحی که ارائه دهنده چیزی تازه هستند، تمرکز داشته باشد. در جهان سخت افزار، استفاده مجدد از قطعات، بخشی طبیعی از فرآیند مهندسی است. در مهندسی نرم افزار این امر به تازگی مورد توجه قرار گرفته است.

یک قطعه نرم افزاری باید چنان طراحی و پیاده سازی شود که بتوان در برنامه های متفاوت از آن بهره برد. در دهۀ ۱۹۶۰، کتابخانه هایی از زیر روال های علمی ساختیم که در آرایهٔ گسترده‌ای از کاربردهای مهندسی و علمی قابل استفاده بودند. این کتابخانه ها از الگوریتم هایی معین به شیوه‌ای کارآمد استفاده می کردند، ولی دامنه کاربرد محدودی داشتند. امروزه، ایدهٔ استفاده مجدد نه تنها الگوریتم ها، بلکه ساختمان داده ها را نیز دربر می گیرد. قطعات مدرن قابل استفاده مجدد، هم داده‌ها و هم پردازشی را که در مورد آنها اعمال می گردد، پنهان سازی کرده مهندس نرم افزار را قادر می سازد تا از قطعات قابل استفادهٔ مجدد، برنامه های کاربردی جدید بسازد. برای مثال، واسطهای کاربر گرافیکی امروزی با استفاده از قطعات قابل استفاده مجدد ساخته می شوند که ایجاد پنجره های گرافیکی، منوهای باز شونده و انواع راهکارهای محاوره را میسر می سازند.

۱-۲-۲ کاربردهای نرم افزار

نرم افزار را در وضعیتی می توان به کار برد که در آن یک مجموعه مراحل از پیش تعیین شده (یعنی یک الگوریتم) تعریف شده باشد (استثنائات قابل ملاحظه در این خصوص، نرم افزارهای سیستم های خبره و نرم افزارهای شبکه عصبی‌اند). محتوای اطلاعاتی و قطعیت اطلاعاتی عوامل مهمی در تعیین ماهیت کاربرد یک نرم افزار هستند. منظور از محتوا، معنی و شکل اطلاعات ورودی و خروجی است. برای مثال، در بسیاری کاربردهای تجاری، از داده های

ورودی بسیار ساخت یافته (یک بانک اطلاعاتی) استفاده می شود و «گزارشهای» فرمت شده تولید می شود. نرم افزاری که یک ماشین خودکار را کنترل می کند (مثلاً کنترل عددی) داده هایی مجزا با ساختاری محدود را می پذیرد و فرمانهایی انفرادی را به توالی برای آن ماشین تولید می کند.

قطعیت اطلاعاتی به معنای قابلیت پیش بینی ترتیب و زمان بندی اطلاعات است. یک برنامه تحلیل مهندسی، داده هایی را می پذیرد که دارای ترتیبی از پیش تعیین شده بوده الگوریتم (های) تحلیلی را بدون وقفه اجرا نموده داده های حاصل را در گزارش یا با قالب گرافیکی تولید می کند. چنین کاربردهایی دارای قطعیت هستند. ولی یک سیستم عامل چند منظوره، ورودی هایی را می پذیرد که دارای محتوای گوناگون و زمان بندی اختیاری هستند؛ الگوریتمهایی را اجرا می کند که توسط شرایط خارجی قابل وقفه‌اند و خروجی تولید می کند که تابعی از محیط و زمان است. کاربردهایی با این ویژگی فاقد عزم هستند.

تعیین گروههای کلی با معنی برای کاربردهای نرم افزار قدری دشوار است. با پیچیده‌تر شدن نرم افزار، مرزهای صریح و روشن، رنگ می بازند. زمینه های زیر را می توان به عنوان گروههای کاربردی مشخص کرد:

نرم افزارهای سیستمی. نرم افزار سیستمی مجموعه‌ای از برنامه هاست که برای سرویس دهی به برنامه های دیگر نوشته شده‌اند. برخی نرم افزارهای سیستمی (مثل کامپایلرها، ویراستارها و برنامه‌های کمکی مدیریت فایل) ساختارهای اطلاعاتی پیچیده ولی قطعیت دارند. برخی برنامه های سیستمی دیگر (نظیر قطعات سیستم عامل، راه اندازها، پردازنده های ارتباط راه دور) مقادیر زیادی از داده های میانی را پردازش می کنند. در هر حال، مشخصه های حیطهٔ نرم افزارهای سیستمی عبارتند از: برهمکنش سنگین با سخت افزار کامپیوتر؛ استفاده سنگین توسط چند کاربر؛ عمل کنونی که مستلزم زمانبندی است؛ مدیریت فرآیند پیچیده و اشتراک منابع؛ ساختمان داده های پیچیده و واسطهای خارجی چندگانه.

نرم افزارهای بلادرنگ. نرم افزاری که رویدادهای جهان واقعی را همانطوری که رخ می دهند، نظارت/ تحلیل/ کنترل می کند، نرم افزار بلادرنگ نامیده می شود. عناصر نرم افزار بلادرنگ عبارتند از یک قطعهٔ جمع آوری کنندهٔ داده ها که اطلاعات را از محیط خارجی جمع آوری و قالب بندی می کند؛ یک قطعه تحلیل کننده که اطلاعات را بنا به نیاز کاربردی انتقال می دهد؛ یک قطعه کنترل/ خروجی که به محیط خارجی پاسخ می دهد و یک قطعه نظارت که همهٔ قطعات دیگر را هماهنگ می کند تا پاسخ بلادرنگ (معمولاً بین یک هزارم ثانیه تا یک ثانیه) برقرار بماند.

نرم افزارهای تجاری. پردازش اطلاعات تجاری گسترده‌ترین زمینهٔ کاربرد نرم افزارها را تشکیل می‌دهد. «سیستمهای مجرد» (مثل لیست حقوق، حسابهای دریافت و پرداخت، موجودی انبار و غیره) به نرم افزارهای سیستم اطلاعات مدیریتی (MIS) تکامل یافته‌اند. این نوع برنامه های کاربردی، داده های موجود را دوباره به شیوه‌ای سازماندهی می کند که عملیات تجاری و تصمیم گیری مدیریتی تسهیل شوند. این نرم افزارها علاوه بر کاربردهای پردازش داده ها، شامل برنامه های کامپیوتری محاوره‌ای (نظیر پردازش تراکنش نقطهٔ فروش) نیز می شود.

نرم افزارهای مهندسی و علمی. نرم افزارهای علمی توسط الگوریتم هایی مشخص می شوند که «ارقام و اعداد» را پردازش می کنند. کاربردهای آن از نجوم تا بررسی آتش فشانها، از تحلیل فشار اتوموتیو تا دینامیک مدار شاتلهای فضایی و از زیست شناسی مولکولی تا مکانیزاسیون صنعتی را دربر می گیرد. ولی، کاربردهای نوین در حیطهٔ مهندسی و علمی از الگوریتم های عددی مرسوم فراتر رفته‌اند. طراحی به کمک کامپیوتر، شبیه سازی سیستم ها، و برنامه های کاربردی محاوره‌ای دیگر، رفته رفته خصوصیات نرم افزارهای بلادرنگ و نرم افزارهای سیستمی را به خود می‌گیرند.

نرم افزارهای تعبیه شده. محصولات هوشمند تقریباً در هر بازار صنعتی و مصرفی جای خود را باز کرده‌اند. نرم افزار تعبیه شده در حافظهٔ فقط خواندنی جای دارد و برای کنترل محصولات و سیستمهای مربوط به بازارهای صنعتی و مصرفی به کار می رود. نرم افزار تعبیه شده قادر به انجام اعمالی بسیار محدود و اختصاصی (از قبیل کنترل صفحه کلید برای فرهای مایکروویو) بوده یا وظایف مهم و قابلیت کنترل (مانند عملیات دیجیتال در خودروها از قبیل کنترل سوخت، صفحه نمایش داشبورد، سیستم ترمز و غیره) را بر عهده دارد.

نرم افزارهای کامپیوترهای شخصی. بازار نرم افزارهای کامپیوتری شخصی طی دو دههٔ اخیر به سرعت رشد یافته است. واژه پردازی، صفحات گسترده، گرافیک کامپیوتری، چند رسانه‌ای، سرگرمی، مدیریت بانکهای اطلاعاتی، برنامه های کاربردی مالی شخصی و تجاری، شبکه خارجی یا دستیابی به بانکهای اطلاعاتی فقط چند مورد از صدها کاربرد در این حیطه است.

نرم افزارهای مبتنی بر وب. صفحات وبی که توسط یک مرورگر بازیابی می شوند، نرم افزارهایی هستند که دستورات اجرایی (مثل CGI، HTML، Perl یا جاوا) و داده هایی (مثل فوق متن و انواع فرمتهای تصویری و صوتی) را به هم مرتبط می سازند. در اصل، شبکه به یک کامپیوتر عظیم تبدیل می شود که یک منبع نرم افزاری تقریباً نامحدود فراهم می آورد؛ منبعی که هر کس با داشتن مودم قادر به دستیابی به آن است.

نرم افزارهای هوش مصنوعی. نرم افزارهای هوش مصنوعی (AI) از الگوریتم های غیر عددی برای حل مسائل پیچیده‌ای که به روشهای عددی قابل حل نیستند، استفاده می کنند. سیستم های خبره، که سیستم های مبتنی بر آگاهی نیز نامیده می شوند؛ تشخیص الگوهای (تصویری و صوتی)؛ شبکه های عصبی مصنوعی؛ اثبات قضایا و بازی، همگی مثالهایی از کاربرد این گروه هستند.

۱-۳ نرم افزار: بحرانی در افق؟

واژه بحران به معنای نقطهٔ عطفی در دوره حیات یک چیز است، مثل زمان یک تصمیم گیری مهم، یک مرحله یا یک رویداد. در مورد کیفیت کلی نرم افزار و سرعت تکامل محصولات و سیستم های کامپیوتری، هیچ نقطهٔ عطفی یا هیچ زمان تصمیم گیری وجود نداشته است، بلکه فقط تغییرات تکاملی آهسته‌ای مشاهده می شود که توسط تغییرات فنّاوری شدید در رشته های مرتبط با نرم افزار دچار وقفه شده است.

۱-٤ اسطوره های نرم افزاری

بسیاری از دردهای نرم افزاری را می توان در اسطوره هایی جست که طی نخستین روزهای تکوین نرم افزارها پدید آمد. برخلاف اسطوره های باستانی که غالباً درس عبرت به بشر می آموزند، اسطوره‌های نرم افزاری باعث اطلاع رسانی نادرست و سردرگمی شده‌اند. اسطوره های نرم افزاری دارای چند ویژگی هستند که آنها را زیانبار ساخته‌اند؛ برای نمونه، به ظاهر بیانی منطقی از واقعیتها بوده‌اند (گاه چند عنصر واقعی در آنها وجود دارد) ولی دارای احساسی نبوغ آمیز بوده غالباً توسط برنامه نویسان کارآزموده‌ای که «قدر می دانند» به آگاهی عموم می رسند.

اسطوره های مدیریتی. مدیرانی که مسئولیت نرم افزاری دارند، همانند مدیران دیگر، غالباً تحت فشار کاهش هزینه ها، جلوگیری از بی برنامگی و بهبود بخشیدن به کیفیت هستند. مدیر نرم افزاری همانند غریقی که به هر

چیزی دست می اندازد، غالباً به اسطوره های نرم افزاری اعتقاد پیدا می کند، اگر بداند این اعتقاد باعث کاهش فشار می شود (حتی به طور موقت).

اسطوره. ما از قبل کتابی داریم که آکنده از استانداردها و روالهای لازم برای ساختن نرم افزارهاست. آیا این، آنچه را که افراد من باید بدانند در اختیارشان قرار نخواهد داد؟

واقعیت. ممکن است کتابِ استانداردهای خیلی خوبی وجود داشته باشد، ولی آیا از آن استفاده می‌شود؟ **آیا سازندگان نرم افزار از وجود آن آگاهند؟** آیا مهندسی نرم افزار نوین را ارائه می دهد؟ آیا کامل است؟ آیا آنقدر روان هست که زمان تحویل را بهبود بخشد و در عین حال کیفیت را حفظ کند؟ در بسیاری از موارد، پاسخِ اکثر این پرسشها «خیر» است.

اسطوره. افراد من ابزارهای نرم افزارسازی حرفه‌ای را دارند؛ به علاوه جدیدترین کامپیوترها را برای آنها خواهیم خرید.

واقعیت. برای ساخت نرم افزارهایی با کیفیت بالا، فقط داشتن آخرین مدل کامپیوتر یا ایستگاه کاری کافی نیست. برای رسیدن به بهره وری و کیفیت خوب، ابزارهای مهندسی نرم افزار به کمک کامپیوتر (CASE) اهمیت بیشتری از سخت افزار دارند، با این وجود، اکثر نرم افزار نویسان باز هم از آنها استفادهٔ مؤثر به عمل نمی آورند.

اسطوره. اگر از برنامه عقب بیفتیم، می توانیم بر تعداد برنامه نویسان بیفزاییم و عقب افتادگی را جبران کنیم (این وضعیت را گاه «horde مغولی» می گویند).

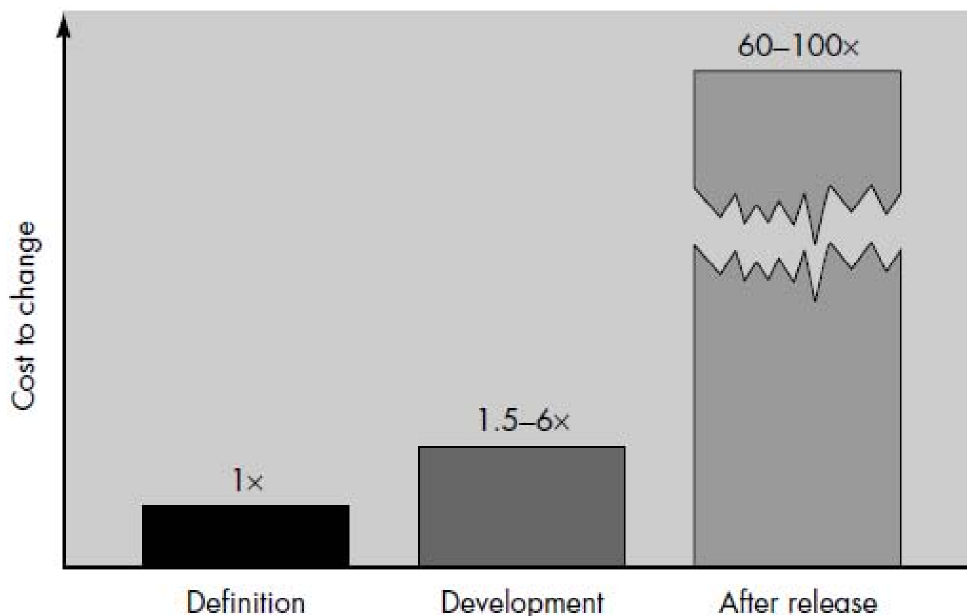
واقعیت. ایجاد نرم افزار، یک فرآیند مکانیکی نظیر ساخت تولیدات معمولی نیست. به قول بروکز [BRO75] : «... با افزودن افراد دست اندرکار به نرم افزاری که تأخیر دارد، بر میزان تأخیر آن افزوده خواهد شد». در نگاه نخست ممکن است این گفته خلاف منطق به نظر برسد، ولی با از راه رسیدن افراد جدید، افراد قدیمی باید زمانی را صرف آموزش آنها کنند و در نتیجه زمانی که باید صرف کار روی نرم افزار شود، هدر می رود. اضافه کردن افراد، عملی است ولی به شیوه‌ای هماهنگ و با برنامه ریزی منظم.

اسطوره های مشتریان. مشتری که درخواست یک نرم افزار کامپیوتری دارد، ممکن است پشت میز کناری باشد، یک گروه تکنیکی در آن سوی سالن باشد، بخش فروش و بازاریابی باشد، یا یک شرکت دیگر باشد که قراردادی برای نرم افزار منعقد نموده است. در بسیاری موارد، مشتری به اسطوره هایی درباره نرم افزارها اعتقاد دارد، زیرا مدیران نرم افزار و سازندگان آن کمتر سعی در برطرف کردن سوءِ تفاهم ها دارند. این اسطوره منجر به انتظارات نادرست (از جانب مشتری) و در نهایت عدم رضایت از سازنده می شود.

اسطوره. نیازهای پروژه پیوسته در حال تغییر است، ولی این تغییرات را به راحتی می توان در نرم افزار جای داد زیرا نرم افزار انعطاف پذیر است.

واقعیت. این درست است که نیازمندی های نرم افزار تغییر می کند ولی تأثیر تغییر به زمان اعمال تغییر بستگی دارد. شکل ۳-۱ اثر تغییر را نشان می دهد. اگر به تعریف صریح توجه جدی شود، درخواستهای اولیه برای تغییر را به راحتی می توان پاسخ گفت. مشتری می تواند نیازمندی ها را مرور کند و اصلاحاتی را با تأثیر نسبتاً کم بر هزینه ها توصیه کند. هنگامی که تغییرات در اثنای طراحی نرم افزار درخواست می شوند، هزینه ها به سرعت بالا می رود. منابع مصرف شده‌اند و یک چارچوب طراحی مشخص شده است. تغییر می تواند باعث تغییرات مشکل آفرینی شود که نیاز به منابع اضافی و اصلاح اساسی طراحی دارد و این یعنی بالا رفتن هزینه ها. تغییرات در عملکرد، کارایی، واسطها، یا

ویژگیهای دیگر در اثنای پیاده سازی (دستورها و آزمایش) اثری شدید بر هزینه دارد. تغییر، در صورتی که نرم افزار به مرحلهٔ استفاده رسید، هزینه ای به مراتب بالاتر خواهد داشت.



شکل ۳-۱ تأثیر تغییر

اسطوره های سازندگان. اسطوره هایی که نرم افزارنویسان باور دارند، نتیجه ۵۰ سال فرهنگ برنامه نویسی است. در نخستین دهه های ساخت نرم افزار، برنامه نویسی شکلی از هنر پنداشته می شد. سنتهای قدیمی دیر از بین می روند.

اسطوره. هنگامی که برنامه را نوشتیم و برنامه کار کرد، دیگر کار تمام است.

واقعیت. یک بار کسی گفته بود: «هر چه زودتر دست به کار نوشتن دستورهای برنامه شوید، زمان بیشتری صرف به پایان بردن آن خواهد کرد». داده های صنعتی [LIE80 , JON91 , PUT97] نشان می دهد که بین ۶۰٪ تا ۸۰٪ از همهٔ کوششهای صرف شده روی نرم افزارها، پس از نخستین بار تحویل آنها به مشتری صورت می پذیرد.

اسطوره. تا هنگامی که برنامه را «اجرا» نکرده‌ام، راهی برای ارزیابی کیفیت آن ندارم.

واقعیت. یکی از مؤثرترین راهکارهای تضمین کیفیت نرم افزار از زمان آغاز پروژه قابل اجراست- یعنی **مرور تکنیکی رسمی**. مرور نرم افزار یک فیلتر کیفیتی است که از آزمایش نرم افزار برای یافتن گروههای معینی از معایب نرم افزاری مؤثرتر است.

اسطوره. تنها چیز قابل تحویل برای یک پروژه موفق برنامه‌ای است که کار کند.

واقعیت. برنامه‌ای که کار می کند فقط بخشی از پیکربندی نرم افزار است که شامل عناصر فراوان می شود. مستندسازی، بنیاد مهندسی موفق بوده و مهمتر از آن راهنمایی برای پشتیبانی نرم افزار فراهم می آورد.

فصل ۲ : فرآیند

فرآیند چیست؟ وقتی که کار می کنید تا یک سیستم یا محصول بسازید، حتماً باید یک سری مراحل قابل پیش بینی را چک کنید یک نقشهٔ راه که در ایجاد نتیجه‌ای با کیفیت بالا و به موقع شما را یاری می کند. این نقشه که آن را دنبال می کنید، «فرآیند نرم افزار» نام دارد.

چه می کند؟ مهندسان نرم افزار و مدیران آنها، فرآیند را با نیازهای خود مطابقت داده سپس آن را دنبال می کنند. به علاوه، کسانی که نرم افزار را درخواست کرده‌اند، در فرآیند نرم افزار نقش دارند.

چرا اهمیت دارد؟ زیرا باعث ثبات، کنترل و سازماندهی فعالیتی می شود که اگر به حال خود گذاشته شود ممکن است باعث آشوب شود.

چه مراحلی دارد؟ در سطح مشروح، فرآیندی که برمی گزینید بستگی به نرم افزاری دارد که می‌خواهید بسازید. یک فرآیند ممکن است برای ایجاد نرم افزار مربوط به سیستم هوا- فضای یک هواپیما مناسب باشد، حال آنکه با ایجاد یک سایت وب ممکن است فرآیندی کاملاً متفاوت مورد نیاز باشد.

حاصل کار چیست؟ از دیدگاه مهندس نرم افزار، حاصل کار، برنامه ها، داده ها و مستنداتی است که به عنوان نتیجه-ای از فعالیتهای مهندسی نرم افزار مشخص شده توسط فرآیند تولید می شوند.

چطور مطمئن شوم که درست از عهده کار برآمده‌ام؟ چند راهکار ارزیابی فرآیند نرم افزار وجود دارد که سازمانها را قادر به تعیین «بلوغ» فرآیند نرم افزار می سازد. ولی کیفیت، به موقع بودن و کارآیی درازمدت محصولی که ساخته‌اید بهترین ملاکها برای بازدهی فرآیند مورد استفاده شما هستند.

ولی دقیقاً یک فرآیند نرم افزار از دیدگاه فنی چیست؟ در این کتاب، فرآیند نرم افزار را به عنوان چارچوبی برای اعمال مورد نیاز جهت ساخت نرم افزار با کیفیت بالا تعریف می کنیم. آیا «فرآیند» مترادف با مهندسی نرم افزار است؟ پاسخ این است: «بلی» و «خیر». فرآیند نرم افزار روش مهندسی را مشخص می کند. ولی مهندسی نرم افزار شامل فنّاوریهایی که فرآیند را تشکیل می دهند- روشهای فنی و ابزارهای خودکار- نیز می شود.

مهمتر اینکه، مهندسی نرم افزار توسط افرادی خلاق و آگاه انجام می شود و باید در چارچوب یک فرآیند نرم افزاری مشخص کار کنند که مناسب محصولات ساختهٔ دست آنها بوده بازار خاص خود را طلب کند. هدف این فصل آن است که وضعیت کنونی فرآیند نرم افزار مورد بررسی قرار گیرد و اشاره هایی نیز به مباحث مدیریتی و فنی بشود که در فصول آتی خواهد آمد.

۲-۱ مهندسی نرم افزار- فنّاوری لایه‌ای

گرچه صدها نویسنده، تعاریفی شخصی از مهندسی نرم افزار ارائه داده‌اند، تعریفی که فریتز باور [NAU69] در یک همایش مهم ارائه کرده است هنوز هم مبنای بحث ما را تشکیل می دهد:

[مهندسی نرم افزار عبارت است از وضع اصول مهندسی بجا و مناسب و استفاده از آنها برای به دست آوردن یک نرم افزار مقرون به صرفه که قابل اطمینان بوده روی ماشینهای واقعی به طرزی کارآمد عمل کند.

تقریباً هر خواننده‌ای وسوسه می شود که نکته‌ای به این تعریف بیفزاید. این تعریف چیز زیادی دربارهٔ جنبه های فنی کیفیت نرم افزار نمی گوید؛ این تعریف به طور مستقیم، نیاز به راضی نمودن مشتری یا تحویل به موقع محصول را مشخص نمی کند و در آن ذکری از اهمیت موازین و استانداردها به عمل نمی آید؛ از اهمیت یک فرآیند بالغ سخن به میان نمی آورد و با این همه، تعریف فریتز باور یک تعریف بنیادی است. «اصول مهندسی مناسب و بجا» کدامند که

در بسط نرم افزار کامپیوتری قابل اجرا هستند؟ چطور می توان نرم افزار «مقرون به صرفه‌ای» ساخت که «قابل اطمینان» باشد؟ برای ایجاد برنامه های کامپیوتری که نه تنها یک ماشین واقعی بلکه «ماشینهای واقعی» متفاوت به طرزی کارآمد عمل کنند، چه چیزهایی لازم است؟ اینها پرسشهایی است که همچنان مهندسان نرم افزار را به مبارزه فرا می خواند.

مهندسی نرم افزار؛ (۱) کاربرد یک روش سیستماتیک، علمی و کمیت پذیر در بسط، راه اندازی و نگهداری نرم افزار؛ یعنی استفاده از مهندسی نرم افزار. (۲) مطالعه روشها به صورت ذکر شده در (۱).

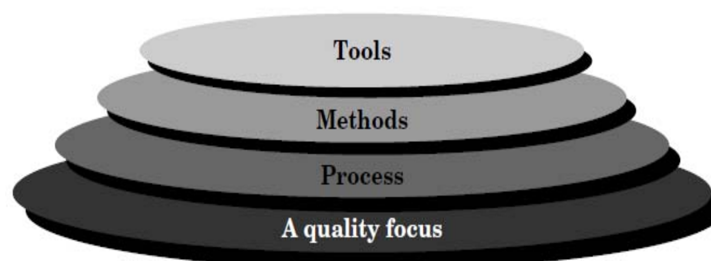
۱-۲-۱ فرآیند، روشها و ابزارها

مهندسی نرم افزار یک فنّاوری لایه‌ای است. شکل ۱-۲ نشان می دهد که هر روش مهندسی (از جمله مهندسی نرم افزار) باید بر کوشش سازمانی در جهت بهبود کیفیت متکی باشد.

بنیاد مهندسی نرم افزار، لایه فرآیند است. مهندسی نرم افزار به مثابه چسبی عمل می کند که لایه‌های فنّاوری را به هم نگه می دارد و بسط موجه و به موقع نرم افزارهای کامپیوتری را میسر می‌سازد. فرآیند، چارچوبی برای یک مجموعه از زمینه های فرآیند کلیدی (KPA) فراهم می آورد [PAU93] که باید برای تحویل مؤثر فنّاوری مهندسی نرم افزار وضع شوند. زمینه های فرآیند کلیدی، پایه‌ای برای کنترل مدیریتی پروژه های نرم افزاری تشکیل داده بستری برای اِعمال روشهای فنی، تولید محصولات کاری (مدلها، مستندات، داده ها، گزارشها، فرمها و غیره)، تعیین مراحل، حصول اطمینان از کیفیت و مدیریت مناسب تغییرات ایجاد می کنند.

روشهای مهندسی نرم افزار، شیوه های فنی برای ساخت نرم افزار را فراهم می آورند. روشها شامل آرایه وسیعی از وظایف از جمله: تحلیل خواسته ها، طراحی، ساخت برنامه ها، آزمایش و پشتیبانی می شوند. روشهای مهندسی نرم افزار متکی بر یک مجموعه اصول بنیادی است که بر تمام زمینه‌های فنّاوری حاکم بوده شامل فعالیتهای مدلسازی و فنون توصیفی دیگر می شوند.

ابزارهای مهندسی نرم افزار، متضمن پشتیبانی خودکار یا نیمه خودکار برای فرآیند و روشها هستند. هنگامی که ابزارها گرد هم آیند به طوری که اطلاعات ایجاد شده توسط یک ابزار، توسط ابزارهای دیگر قابل استفاده باشند، سیستمی برای پشتیبانی بسط نرم افزار شکل می گیرد که مهندسی نرم افزار به کمک کامپیوتر (CASE) نام دارد.



شکل ۱-۲ لایه های مهندسی نرم افزار

۲-۱-۲ دیدی کلی از مهندسی نرم افزار

مهندسی عبارت از تحلیل، طراحی، ساخت، بررسی و مدیریت نهادهای فنی (یا اجتماعی) است. نهاد مورد نظر هرچه که باشد، پرسشهای زیر باید پاسخ داده شوند:

- مسئله‌ای که باید حل شود، کدام است؟
- کدام ویژگیهای نهاد هستند که برای حل مسئله به کار برده می شوند؟
- نهاد (و راه حل) چگونه درک می شود؟
- نهاد چگونه بنا خواهد شد؟
- از چه روشی برای کشف خطاهای صورت پذیرفته در طراحی و ساخت نهاد استفاده خواهد شد؟
- نهاد در دراز مدت و آن هنگام که کاربران نهاد درخواست تغییراتی در جهت تصحیح و بهبود آن را دارند، چگونه مورد پشتیبانی قرار خواهد گرفت؟

در سرتاسر این کتاب تنها یک نهاد مدنظر ما خواهد بود و آن نرم افزار کامپیوتری است. برای مهندسی مناسب نرم افزار، باید یک فرآیند مهندسی نرم افزار تعریف گردد. در این بخش به ویژگیهای کلی فرآیند نرم افزار خواهیم پرداخت بعداً در همین فصل، مدلهای فرآیند مشخص را بررسی خواهیم نمود.

کار مرتبط با مهندسی نرم افزار را در سه فاز کلی می توان گروه بندی کرد. این موضوع به حیطهٔ برنامه کاربردی، اندازهٔ پروژه یا پیچیدگی آن مربوط نمی شود. هر فاز با یک یا چند پرسش مطرح شده در بالا سروکار دارد.

فاز تعریف، بر چیستی تأکید دارد. یعنی طی فاز تعریف، مهندس نرم افزار می کوشد تا تعیین کند چه اطلاعاتی باید پردازش شود، کدام عمل و کارآیی مطلوب است، چه رفتارهای سیستمی قابل انتظار است، چه رابطه هایی را می توان برقرار نمود، چه محدودیتهایی در طراحی وجود دارد، و چه ملاکهای معتبرسازی برای تعریف یک سیستم موفق مورد نیاز است؟ خواسته های کلیدی سیستم و نرم افزار تشخیص داده می شود. گرچه روشهای اعمال شده طی فاز تعریف، به الگوی (یا ترکیبی از الگوهای) مهندسی نرم افزار بستگی دارند، سه کار عمده به نحوی صورت می پذیرد: مهندسی اطلاعات یا سیستم، طرح ریزی پروژه نرم افزار و تحلیل خواسته ها.

فاز توسعه، بر چگونگی تأکید دارد، یعنی در اثنای توسعه، مهندس نرم افزار می کوشد تعیین کند داده ها چه ساختاری داشته باشند، عملیات چگونه در یک معماری نرم افزاری پیاده سازی شوند، جزئیات روال ها چگونه پیاده سازی شوند، واسطها چگونه ویژگیهایی داشته باشند، طراحی چگونه به یک زبان برنامه نویسی (یا یک زبان غیر روالی) ترجمه شود و آزمایش به چه نحو انجام شود. روشهای اعمال شده طی فاز توسعه متغیرند، ولی سه وظیفهٔ فنی همواره باید به انجام برسد: طراحی نرم افزار، تولید دستورها و آزمایش نرم افزار.

فاز پشتیبانی بر تغییراتی تأکید دارد که با تصحیحات مورد نیاز در جهت تکامل محیط نرم افزار در ارتباط هستند و نیز تغییراتی که ناشی از تغییر خواسته های مشتریان هستند. طی فاز پشتیبانی چهار نوع تغییر قابل مشاهده است:

تصحیح: حتی در صورت انجام بهترین کارها برای حصول اطمینان از کیفیت، باز هم این احتمال وجود دارد که مشتری معایبی را در نرم افزار بیابد. نگهداری تصحیحی، نرم افزار را در جهت تصحیح معایب تغییر می دهد.

تطابق: به مرور زمان، محیط اولیه (مثل CPU، سیستم عامل، قواعد کار، ویژگیهای محصول خارجی) که نرم افزار برای آنها بسط یافته است، احتمالاً تغییر می کند. نگهداری تطابقی منجر به انجام اصلاحاتی در نرم افزار می شود به نحوی که پاسخگوی تغییرات محیط خارجی خود شود.

بهبود: به موازاتی که نرم افزار مورد استفاده قرار می گیرد، مشتری/ کاربر متوجه عملکردهایی می‌شوند که افزودن آنها موجب بهتر شدن نرم افزار می شود. نگهداری بهبودی نرم افزار را به فراسوی خواسته های اولیه آنها سوق می دهد.

جلوگیری: نرم افزار کامپیوتری در اثر تغییرات زیاد، کارایی خود را از دست می دهد و از اینرو، نگهداری پیشگیرانه که غالباً مهندسی مجدد نرم افزار نامیده می شود، باید اجرا شود تا نرم افزار نیازهای کاربران نهایی خود را برآورده سازد. در اصل، نگهداری پیشگیرانه تغییراتی در برنامه های کامپیوتر اعمال می کند که بتوان آنها را راحت‌تر تصحیح کرد، تطابق داد و بهبود بخشید.

این فازها و مراحل مرتبطی که در دید کلی ما از مهندسی نرم افزار شرح داده شده با چند فعالیت پوششی تکمیل می شوند. اعمال متداول در این گروه عبارتند از:

- کنترل و دنبال کردن پروژه نرم افزاری
- مرور فنی رسمی
- تضمین کیفیت نرم افزار
- مدیریت پیکربندی نرم افزار
- تولید و تهیه مستندات
- مدیریت قابلیت استفاده مجدد
- سنجش
- مدیریت ریسک

۲-۲ فرآیند نرم افزار

فرآیند نرم افزار را می توان مانند شکل ۳-۲ مشخص کرد یک چارچوب فرآیند مشترک با تعریف تعداد کمی از اعمال چارچوب بنا می شود که در همه پروژه های نرم افزار (بدون توجه به اندازه و میزان پیچیدگی آنها) قابل اجرا هستند. چند مجموعه ای از کارهای عملی در مهندسی نرم افزار، نقاط عطف پروژه، قطعات قابل تحویل و نکات تضمین کیفیت است تطبیق اعمال چارچوب از قبیل تضمین کیفیت نرم افزار، مدیریت پیکربندی نرم افزار و سنجش روی مدل فرآیند قرار می گیرد فعالیت های پوششی، مستقل از اعمال چارچوب هستند و در سرتاسر فرآیند رخ می دهند.

در سال های اخیر، تاکید فراوانی بر «بلوغ فرآیند» شده است موسسه مهندسی نرم افزار (SEI) یک مدل مفهومی پی ریزی کرده است که مبتنی بر مجموعه ای از قابلیت های مهندسی نرم افزار است که باید به موازات رسیدن سازمان به سطوح متفاوتی از بلوغ فرآیند، ارائه شوند برای تعیین حالت کنونی یک سازمان از لحاظ بلوغ فرآیند SEI از یک ارزیابی استفاده می کندکه منجر به پنج الگوی امتیازبندی می شود. الگوی امتیازبندی مطابقت با یک مدل بلوغ قابلیت را تعیین می کند روش SEI میزانی از تاثیر جهانی فعالیت های یک شرکت در زمینه مهندسی نرم افزار ارائه کرده پنج سطح بلوغ فرآیند را بنا می کند که به شیوه زیر تعریف می شوند:

سطح۱: آغاز. فرآیند نرم افزار به صورتی خاص و حتی گاه به صورتی آشفته مشخص می شود چند فرآیند تعریف می شود و موفقیت به کوشش های فردی بستگی دارد.

سطح۲: تکرار پذیر. فرآیندهای اصلی مدیریت برای کنترل هزینه ها، زمانبندی و عملکرد ایجاد می شوند انضباط در فرآیند برای تکرار موفقیت های قبلی روی پروژه هایی با کاربرد مشابه ضروری است.

سطح۳: تعریف شده. فرآیند نرم افزار برای هر دو فعالیت مدیریتی و مهندسی، مستندسازی و استاندارد شده در قالب یک فرآیند نرم افزار درون سازمانی ارائه می شود همه پروژها از یک نسخه مستند و به تصویب رسیده از فرآیند سازمان برای توسعه و پشتیبانی نرم افزار استفاده می کنند این سطوح تمام ویژگی های تعریف شده برای سطح۲ را در بر می گیرد.

سطح٤: مدیریت شده. موازین مفصلی از فرآیند نرم افزار و کیفیت محصول جمع آوری می شود هم فرآیند نرم افزار و هم محصولات از لحاظ کمی درک می شوند و با استفاده از این موازین مفصل کنترل می شوند این سطح کلیه ویژگی های تعریف شده برای سطح ۳ را در برمی گیرد.

سطح ۵: بهینه سازی. بهبود پیوسته فرآیند از طریق بازخورد کمی فرآیند، و از آزمون ایده ها و فناوری های جدید میسر می شود این سطح شامل همه ویژگی های تعریف شده برای سطح ٤ می شود.

لازم به تاکید است که هر یک از سطوح۲ به بالا، سطح قبلی خود را در بر می گیرد، برای مثال نرم افزارنویسی که در سطح۲ قرار دارد باید به وضعیت سطح۲ رسیده باشد تا بتواند به فرآیندهای ذکر شده در سطح ۳ دست پیدا کند.

در سطح ۱ کوشش چندانی روی تضمین کیفیت و مدیریت پروژه صورت نمی پذیرد و در آن هر تیم پروژه ای مجاز به ایجاد نرم افزار به هر شیوه و با استفاده از هر گونه روش، استاندارد و رویه ای است که ممکن است از خوب تا بسیار ضعیف در تغییر باشند.

سطح۲: نشانگر این واقعیت است که یک نرم افزار نویس، عملیات معینی از قبیل گزارش اتمام کار و گزارش زمان و کار صرف شده را تعیین کرده است.

سطح۳ نشانگر این واقعیت را نشان می دهد که نرم افزار نویس هم فرآیندهای مدیریتی و هم فنی را تعیین کرده است، برای مثال، استانداردی برای برنامه نویس وضع شده است و توسط روال هایی نظیر امور بازرسی تقویت می شود در این سطح است که اکثر نرم افزار نویس ها اقدام به اخذ استانداردهایی از قبیل ISO 9001 می کنند تعداد نرم افزار نویسانی که از این سطح فراتر بروند بسیار اندک است.

سطح٤ در برگیرنده مفهوم میزان و استفاده از استانداردهاست در فصل ٤ این مبحث به تفضیل مورد بحث قرار خواهد گرفت ولی طرح مفهوم معیار به منظور درک اهمیت دستیابی یک نرم افزار نویس به سطح٤ یا ۵ ارزشمند است.

معیار، یک کمیت با معنی است که با آن می توان از یک سند یا کد موجود در پروژه، نرم افزاری را استخراج نمود، مثالی از چنین معیارهایی، تعداد انشعاب های شرطی در بخشی از کد برنامه است این معیار از آن رو مفید است که تا حدی کوشش لازم برای آزمون کد را نشان می دهد یعنی با تعداد مسیرهای آزمون موجود در کد، ارتباط مستقیم دارد.

سازمانی که در سطح ٤ قرار دارد معیارهای متعددی جمع آوری می کند سپس این معیارها برای نظارت بر پروژه و کنترل آن مورد استفاده قرار می گیرد، برای مثال:

- ممکن است از یک معیار آزمون برای تعیین زمان به پایان رساندن آزمایش بخشی از کد استفاده شود.
- از یک معیارخوانایی برای قضاوت درباره خوانا بودن یک سند به زبانی طبیعی استفاده شود.

- از یک معیار فهم برنامه برای تعیین یک آستانه عددی استفاده شود که برنامه نویسان مجاز به عدول از آن نیستند.

برای آنکه معیارها در این سطح موفق باشند، لازم است، تمام اجزای سطح ۳ برقرار باشند، برای مثال، نهادهای مشخصی برای اعمال نظیر مشخص کردن خواسته ها و طراحی وجود دارد به طوری که معیار به سهولت توسط ابزارهای مکانیکی قابل استخراج باشند.

سطح۵، بالاترین سطح قابل دستیابی است هنوز تعداد بسیار ناچیزی از سازندگان نرم افزار به این مرحله رسیده اند در این سطح همتای نرم افزار، راهکارها، کنترل کیفیت که هنوز هم در صنایع بالغتر وجود دارند به چشم می خورد برای مثال، تولید کننده یک محصول صنعتی مانند بلبرینگ قادر به نظارت و کنترل کیفیت بلبرینگ های تولید شده و قادر به پیش بینی کیفیت براساس ماشین ها و فرآیندهای تولیدی به کار رفته در ساخت بلبرینگ هاست، به همین ترتیب، سازنده نرم افزاری که در سطح ۵ قرار دارد، قادر به پیش بینی پیامدهایی از قبیل تعداد اشکال های باقی مانده در یک محصول براساس سنجش های انجام شده در حین اجرای پروژه است، به علاوه، چنین سازنده ای قادر است اثر کمی یک فرآیند یا ابزار تولید جدید روی پروژه را تعیین کند برای این منظور وی، معیارهای آن پروژه را مورد بررسی قرار داده آنها را با پروژه های قبلی که از آن فرآیند یا ابزار استفاده نمی کرده اند مقایسه می کند.

تاکید بر این نکته ضروری است که برای آنکه سازنده نرم افزاری به سطح ۵ برسد باید به طور دقیق هر فرآیند را مشخص کرده باشد و آن را بدون کوچکترین تغییری دنبال کند این نتیجه ای است که از سطح ۳ عاید می شود اگر سازنده نرم افزاری، فرآیندها را به طور دقیق مشخص نکرده باشد افت و خیز زیادی در فرآیند ساخت وجود خواهد داشت و از آماری که برای اعمالی از قبیل پیش بینی به کار رفته اند، نمی توان بهره برد.

پنج سطحی که SEI تعیین کرده است نتیجه پاسخ به پرسشنامه ارزشیابی SEI است که مبتنی بر CMM می باشد نتایج این پرسشنامه به یک نمره عددی خلاصه می شود که نشانگر رشد و بلوغ فرآیند در یک سازمان است SEI، به هر سطح بلوغ یک سری زمینه های فرآیند کلیدی (KPA) نسبت داده است KPA ها آن دسته از وظایف مهندسی نرم افزار (مانند طرح ریزی پروژه نرم افزاری، مدیریت خواسته ها) را توصیف می کند که باید برای رسیدن به حد مطلوبی از عملکرد در یک سطح معین، به انجام برسند، هر KPA با شناسایی ویژگی های زیر توصیف می شود:

- **اهداف.** مقاصد کلی که KPA باید به آنها برسد.
- **وظایف.** خواسته های (تحمیل شده برسازمان) که باید برای دستیابی به اهداف رعایت شوند، تاییدی بر همخوانی اهداف باشند.
- **توانایی ها**، آن چیزهایی که باید در جای خود (از نظر سازمانی و از نظر فنی) قرار داشته باشند تا سازمان قادر به انجام وظایف خود باشد.
- **روش های نظارت بر پیاده سازی.** شیوه نظارت بر فعالیتها به موازاتی که در جای خود قرار می گیرند.
- **روش های بازبینی پیاده سازی.** شیوه بازبینی مناسب بودن عملی برای KPA.

هیجدهKPA (که همگی با ساختار ذکر شده در بالا توصیف می شوند) در مدل بلوغ قابل تعریف است که در سطوح متفاوتی از بلوغ فرآیند تصویر می شوند، در هر یک از سطوح بلوغ زیر، به KPA های مربوطه باید دست پیدا کرد :

سطح۲بلوغ فرآیند

- مدیریت پیکربندی نرم افزار
- تضمین کیفیت نرم افزار

- مدیریت قرار دادهای نرم افزاری
- کنترل پروژه نرم افزاری و خطای ناخواسته
- طرح ریزی پروژه نرم افزار
- مدیریت خواسته ها

سطح ۳ بلوغ فرآیند

- بازبینی های نظیر
- هماهنگی گروه ها
- مهندسی محصول نرم افزار
- مدیریت یکپارچه نرم افزار
- برنامه آموزش
- تعریف فرآیند سازمانی
- تاکید برفرآیند سازمانی

سطح٤ بلوغ فرآیند

- مدیریت کیفیت نرم افزار
- مدیریت فرآیندهای کمی

سطح٥ بلوغ فرآیند

- مدیریت تغییر فرآیند
- مدیریت تغییر فناوری
- پیشگیری نقایص

هر یک از این KPA ها توسط یک مجموعه کارهای کلیدی تعریف می شود که در رسیدن به اهدافی که دارد سهیم هستند این کارهای کلیدی عبارتند از سیاست ها، روال ها و فعالیت هایی که باید پیش از نهادینه شدن کامل یک زمینه فرآیند کلیدی، به انجام برسند، SEI یک سری نشانه های کلیدی به عنوان کارهای کلیدی یا مولفه هایی از کارهای کلیدی تعریف می کند که به بهترین وجه نشان می دهند آیا اهداف یک زمینه فرآیند کلیدی برآورده شده است یا خیر. برای آزمودن وجود (یا نبود) یک نشانه کلیدی، پرسش های ارزیابی طراحی شده است.

۳-۲ مدل های فرآیند نرم افزار

برای حل مسائل واقعی در یک مجموعه صنعتی، یک مهندس نرم افزار یا تیمی از مهندسان باید یک راهبرد توسعه تعیین کنند که در برگیرنده لایه های فرآیند، روش ها و ابزارها (که در بخش ۱-۱-۲ شرح داده شد) و فازهای کلی شرح داده شده در بخش ۲-۱-۲ باشد این راهبرد را غالبا مدل فرآیند یا الگوی مهندسی نرم افزار می نامند یک مدل فرآیند برای مهندسی نرم افزار براساس ماهیت پروژه و نوع کاربرد، روش ها و ابزارهای مورد استفاده و کنترل ها و قطعات قابل تحویل موردنیاز انتخاب می شود.

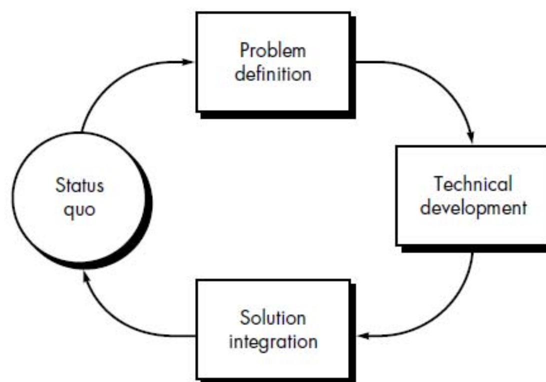
توسعه نرم افزار را در کل می توان به عنوان یک حلقه حل مسئله در نظر گرفت(شکل ۳-۲ الف) که در آن چهار مرحله متمایز به چشم می خورد، وضع موجود، تعیین مسئله، توسعه فنی، و جامعیت راه حل. وضع موجود، وضعیت کنونی امور را نشان می دهد؛ تعیین مسئله مشخص می کند که چه مسئله خاصی باید حل شود. توسعه فنی، مسئله را از طریق به کارگیری فناوری حل می کند و با جامعیت راه حل، نتایج(مانند مستندات، برنامه ها، داده ها، وظایف کاری

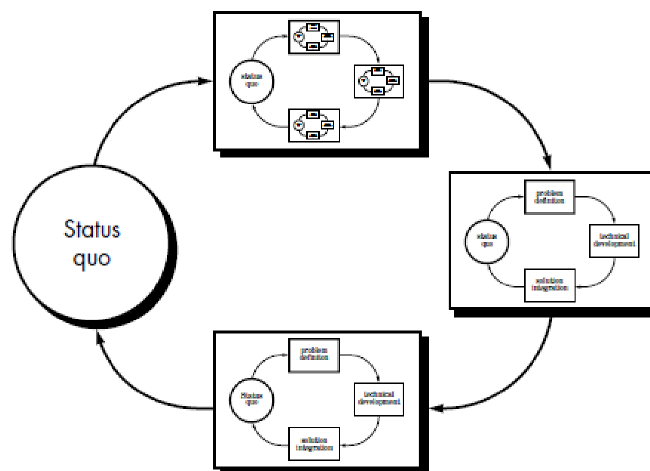
در جهان واقعی، مرزبندی میان فعالیت ها به صورت ایده آلی که در شکل ۳-۲ ب نشان داده شده است. دشوار است زیرا بین مراحل تداخل رخ می دهد همین دید ساده هم منجر به ایده های بسیار مهمی می شود مدل فرآیند انتخاب شده برای یک محصول نرم افزاری، هر چه باشد همه مراحل فوق وضع موجود، تعیین مسئله، توسعه فنی همزمان از یک درجه اهمیت برخوردار هستند. نظر به ماهیت بازگشتی شکل ۳-۲ ب چهار مرحله شرح داده شده در بالا به یک میزان در تحلیل یک کاربرد کامل و تولید قطعه ای کوچک از کد به اجرا گذاشته می شوند.

FIGURE 2.3

(a) The phases of a problem solving loop [RAC95]

(b) The phases within phases of the problem solving loop [RAC95]





شکل ۳-۲

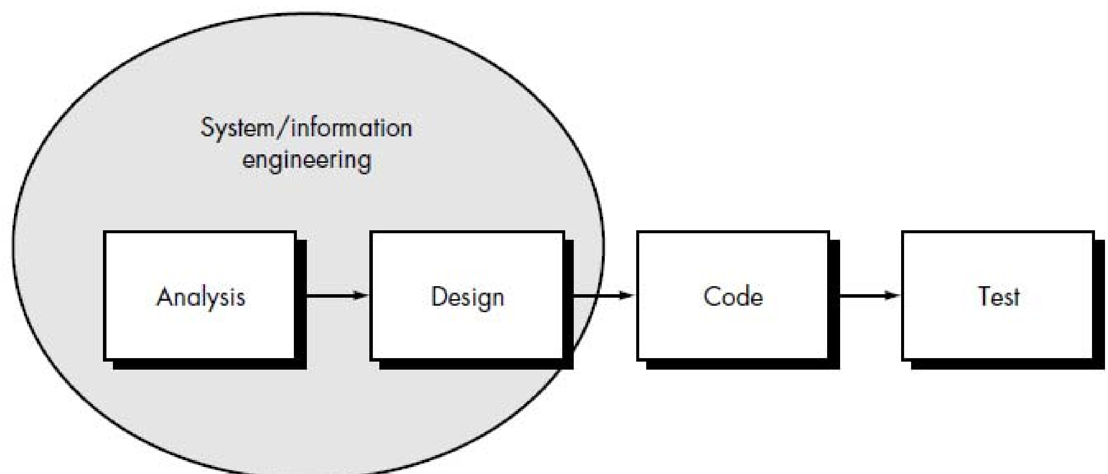
الف) فازهای یک حلقه حل مسئله [RAC95]. ب) فازهای موجود در داخل فازهای حلقه حل مسئله [RAC95].

راکون [RAC95] یک مدل آشوب پیشنهاد می کند که «بسط نرم افزار» را به عنوان طیفی گسترده از کاربر به سازنده و از سازنده به فناوری توصیف می کند با پیشرفت کار به سمت یک سیستم کامل مراحل شرح داده شده در بالا به طور بازگشتی در مورد نیازهای کاربر و مشخصات فنی سازنده نرم افزار به اجرا گذاشته می شود.
در بخش های بعدی، چند مدل فرآیند متفاوت برای مهندسی نرم افزار مورد بحث قرار خواهد گرفت هر یک از این مدل ها کوششی برای به نظم درآوردن یک فعالیت ذاتا آشوبی را به تصویر می کشد ضمنا به خاطر داشته باشید که هر یک از این مدل ها به شیوه ای تعریف شده است که (انشاالله) به کنترل و هماهنگ سازی یک پروژه نرم افزاری واقعی کمک کند با این همه، این مدل ها ویژگی های مدل آشوب را از خود نشان می دهند.

۲-٤ مدل ترتیبی خطی

این مدل که گاه «مدل آبشار» یا «چرخه حیات کلاسیک» نیز خوانده می شود، یک روش سیستماتیک و ترتیبی برای بسط نرم افزار پیشنهاد می کند که در سطح سیستمی آغاز می شود و به تحلیل، طراحی، کدنویسی، آزمایش و پشتیبانی پیشروی می کند شکل ٤-۲ مدل ترتیبی خطی برای مهندسی نرم افزار را نشان می دهد مدل ترتیبی خطی که از روی چرخه مهندسی سنتی گرفته شده است، شامل فعالیت های زیر می شود.

مهندسی سیستم/ اطلاعات و مدل سازی: از آنجا که نرم افزار همواره بخشی از یک سیستم (یا یک مقوله کاری) بزرگتر است، کار با تعیین خواسته های مربوط به همه عناصر سیسستم و سپس اختصاص دادن زیر مجموعه ای از این خواسته ها به نرم افزار آغاز می شود این دید سیستمی هنگامی اهمیت می یابد که قرار است نرم افزار با عناصر دیگری از قبیل سخت افزار، افراد و بانک های اطلاعاتی تعامل داشته باشد تحلیل و مهندسی سیستم شامل جمع آوری خواسته ها در سطح سیسستم، با مقدار اندکی تحلیل و طراحی سطح بالا است، مهندسی اطلاعات شامل جمع آوری خواسته ها در سطح تجارت راهبردی و در سطح زمینه کاری است.



شکل ٤-۲ مدل ترتیبی خطی

تحلیل خواسته های نرم افزار. فرآیند جمع آوری خواسته ها به طرزی خاص بر روی نرم افزار تشدید و متمرکز می شود برای درک ماهیت برنامه (هایی) که باید ساخته شوند، مهندس یا تحلیلگر نرم افزار باید دامنه اطلاعات را برای نرم افزار و همچنین عمل مورد نیاز، رفتار، کارآیی و ایجاد واسط، درک کند.

طراحی. طراحی نرم افزار فرآیندی چند مرحله ای است که بر چهار صفت متمایز از یک برنامه تمرکز دارد، ساختمان داده ها، معماری نرم افزار، نمایش واسط ها و جزئیات رویه ای (الگوریتم ها). فرآیند طراحی، خواسته ها را به نمودی از نرم افزار ترجمه می کند که قبل از کدنویسی می توان از آن برای ارزیابی کیفی استفاده کرد.

تولید کد. طراحی باید به یک شکل قابل فهم برای ماشین تبدیل شود در مرحله تولید کد این وظیفه انجام می شود اگر طراحی به تفضیل صورت پذیرد، تولید کد را می توان به طور خودکار انجام داد.

آزمایش. هنگامی که کدها تولید شدند آزمایش برنامه آغاز می شود فرآیند آزمایش، بر منطق داخلی نرم افزار متمرکز می شود تا اطمینان حاصل شود که همه دستورها مورد آزمایش قرار گرفته اند علاوه بر این، بر اجزای خارجی عملیاتی نیز متمرکز می شود، یعنی آزمون هایی برای کشف خطاها و حصول اطمینان از این که ورودی تعیین شده نتایجی واقعی تولید می کند که با نتایج مورد نیاز همساز است، انجام می گیرد.

پشتیبانی. بدون شک پس از آنکه نرم افزار تحویل مشتری شد، دستخوش تغییر می شود (البته نرم افزارهای تعبیه شده از این قاعده مستثنی هستند). تغییرات از آن رو رخ می دهند که خطاهایی به چشم می خورد، زیرا نرم افزار باید برای سازش با تغییرات محیط خارجی (از قبیل تغییرات لازم به سبب وارد شدن یک سیستم عامل جدید و یا یک دستگاه جانبی جدید) خود را مطابقت دهد، یا این که مشتری نیاز به بهتر کردن عملیات و کارایی دارد پشتیبانی و نگهداری نرم افزار، هر یک از فازهای ذکر شده در بالا را در مورد یک برنامه موجود دوباره به کار می گیرد نه در مورد یک برنامه جدید.

مدل ترتیبی خطی، قدیمی ترین و پرکاربرد ترین الگو برای مهندسی نرم افزار است، ولی نقد این الگو باعث شده که حتی هواداران فعال آن نیز بازدهی آن را مورد تردید قرار دهند. [HAN95]. از جمله مشکلاتی که به هنگام اجرای مدل ترتیبی خطی پیش می آید، می توان به موارد زیر اشاره کرد:

۱. پروژه های واقعی به ندرت جریان ترتیبی پیشنهاد شده توسط این مدل را دنبال می کنند. اگر چه مدل خطی می تواند پذیرای تکرار باشد، این عمل را به طور غیر مستقیم انجام می دهد. در نتیجه، با پیش رفتن تیم پروژه ممکن است تغییرات باعث ایجاد سر در گمی شوند.

۲. غالبا برای مشتری دشوار است که همه نیازهای خود را به وضوح بیان کند، مدل ترتیبی خطی، به بیان واضح نیاز دارد و به خوبی از پس موارد غیر قطعی که در آغاز اکثر پروژه ها وجود دارند بر نمی آید.

۳. مشتری باید حوصله داشته باشد یک نسخه کاری از برنامه ها تا آخرین روز های پروژه در دسترس او قرار نخواهد گرفت یک اشتباه عمده که تا زمان بازبینی برنامه کاری از دید، پنهان بماند می تواند بسیار دردسر آفرین باشد.

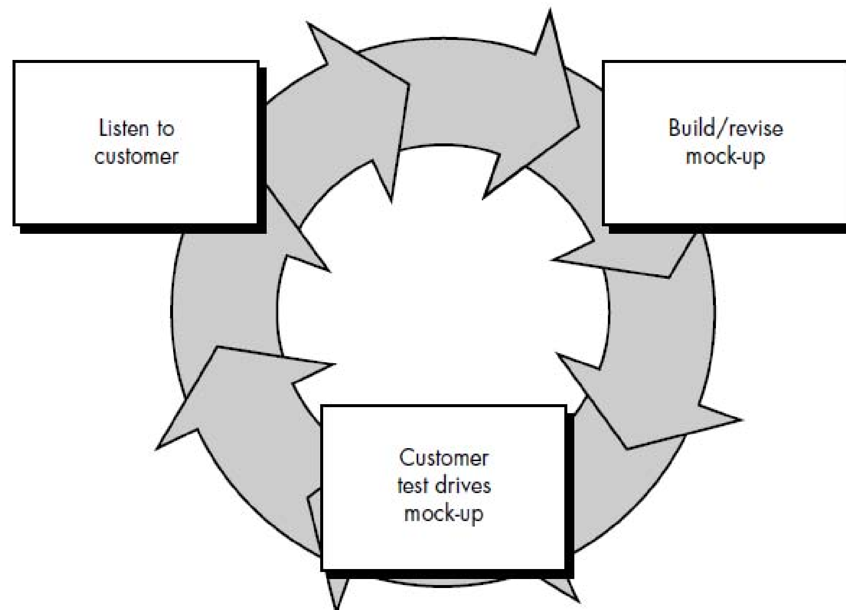
همه این مشکلات جدی هستند، ولی الگوی چرخه حیات کلاسیک جایگاهی مشخص و مهم در کار مهندسی نرم افزار دارد الگویی برای قرار دادن روش های تحلیل، طراحی، کدنویسی، آزمایش و پشتیبانی ارائه می کند چرخه حیات کلاسیک همچنان پر کاربردترین مدل رویه ای برای مهندسی نرم افزار باقی می ماند با وجود نقاط ضعف، به مراتب بهتر از یک روش اتفاقی برای ساخت نرم افزار است.

۲-۵ مدل ساخت نمونه اولیه

غالبا مشتری، یک مجموعه اهداف برای نرم افزار تعیین می کند، ولی جزئیات ورودی ها، پردازش و یا خواسته های خروجی را مشخص نمی کند در موارد دیگر، ممکن است سازنده از بازدهی یک الگوریتم، قابلیت تطابق

الگوی ساخت نمونه اولیه، (شکل ۵-۲) با جمع آوری خواسته ها آغاز می شود مشتری و سازنده با هم ملاقات می کنند و اهداف کلی نرم افزار را تعیین می کنند همه خواسته های معلوم را شناسایی می کنند و زمینه هایی را مطرح می کنند که تعریف بیشتر در آنها الزامی است سپس یک «طراحی سریع» صورت می پذیرد در طراحی سریع، هدف اصلی، ارائه آن دسته از ویژگی های نرم افزار است که به چشم مشتری، کاربرد می آیند (مثل روش های وارد کردن اطلاعات و فرمت های خروجی) طراحی سریع منجر به ساخت یک نمونه اولیه می شود، نمونه اولیه مورد ارزیابی مشتری/ کاربر قرار گرفته و از آن برای پالایش خواسته های نرم افزاری استفاده می شود که قرار است ساخته شود. با تنظیم نمونه اولیه برای برآوردن نیازهای مشتری، تکرار رخ می دهد و در عین حال، سازنده بهتر می فهمد که چه نیازهایی باید برآورده شود.

در حالت ایده آل، نمونه اولیه به عنوان راهکاری برای تشخیص خواسته های نرم افزار عمل می کند، اگر یک نمونه اولیه کاری ساخته شود سازنده می کوشد تا از قطعات برنامه موجود استفاده کند یا از ابزارهایی (مانند مولد گزارش، مدیریت پنجره و غیره) استفاده کند تا برنامه های کاری به سرعت تولید شود.



شکل ۵-۲ الگوی ساخت نمونه اولیه

ولی هنگامی که نمونه اولیه، اهداف ذکر شده در بالا را بر آورده ساخت، با آن چه می کنیم؟ بروکز [BRO75] چنین پاسخ می دهد:

در اکثر پروژه ها، نخستین سیستمی که ساخته می شود چندان قابل استفاده نیست، ممکن است بیش از حد آهسته باشد بیش از حد بزرگ باشد، استفاده از آن دشوار باشد یا اینکه هر سه عیب را با هم داشته باشد چاره ای جز شروع دوباره وجود ندارد باید نسخه دیگری ساخت که این مشکلات در آن حل شده باشد.. هنگامی که یک فناوری جدید یا یک مفهوم سیستمی جدید به کار می رود، باید سیستمی برای دور انداختن درست کرد، زیرا حتی بهترین طراحی ها

نمونه اولیه می تواند به عنوان نخستین سیستم عمل کند، یعنی همان طور که بروکز توصیه می کند، دور انداخته شود، ولی این ممکن است یک دید ایده آل باشد این درست است که هم مشتریان و هم سازندگان، الگوی ساخت نمونه اولیه را دوست دارند کاربران احساس می کنند که یک سیستم واقعی را آزمایش می کنند و سازندگان چیزی را بلافاصله ساخته اند با این همه، ساخت نمونه اولیه نیز می تواند به دلایل زیر مشکل آفرین باشد:

۱. مشتری چیری را می بیند که ظاهرا یک نسخه کاری از نرم افزار است ولی نمی داند که این نمونه اولیه با موم سر هم بندی شده است نمی داند که به لحاظ شتابی که به کارگیری داشته ایم، کیفیت کلی نرم افزار و قابلیت نگهداری دراز مدت مدنظر نبوده است هنگامی که مطلع می شود محصول باید بازسازی شود تا به سطوح بالای کیفیت برسد از کوره در می رود و تقاضا می کند با چند ترمیم جزئی این نمونه اولیه به یک محصول کاری تبدیل شود، اکثر اوقات هم مدیریتی ساخت نرم افزارکوتاه می آید.

۲. سازندگان غالبا برای به کارگیری هر چه سریعتر نمونه اولیه، در پیاده سازی آن کوتاه می آیند، ممکن است از یک سیستم عامل یا زبان برنامه نویسی نامناسب استفاده شود صرفا به خاطر این که در دسترس و شناخته شده است ممکن است یک الگوریتم ناکارآمد پیاده سازی شود صرفا برای آنکه قابلیت برنامه نشان داده شود، پس از مدتی برنامه نویس ممکن است با این انتخاب ها مانوس شود و کلا فراموش کند که چرا مناسب بوده اند انتخاب « کمتر از ایده آل» اکنون به بخشی از سیستم تبدیل شده است.

ممکن است مشکلاتی رخ دهد ولی ساخت نمونه اولیه می تواند الگوی موثری برای مهندسی نرم افزار باشد کیلوکار، تعیین قواعد بازی در همان آغاز است، یعنی مشتری و سازنده هر دو باید بپذیرند که نمونه اولیه بدین منظور ساخته می شود تا به عنوان راهکارهای برای تعیین خواسته ها عمل کند.

۲-۶ مدل RAD

توسعه کاربردی سریع (RAD) یک مدل فرآیند توسعه تدریجی نرم افزار است که بر یک چرخه توسعه بی اندازه کوتاه تاکید دارد، مدل RAD شکل پرسرعت مدل ترتیبی خطی است که در آن بر توسعه سریع با استفاده از ساخت مبتنی بر مولفه ها عمل می شود اگر خواسته ها به خوبی درک شده باشند و دامنه پروژه محدود باشد، فرآیند RAD تیم سازنده را قادر می سازد تا یک سیستم کاملا عملیاتی در مدت زمانی بسیار کوتاه (مثلا بین ۶۰ تا۹۰ روز) ایجاد کند. روش RAD که نخستین بار برای کاربردهای سیستم اطلاعات مورد استفاده قرار گرفت، شامل فازهای زیر می شود:

مدلسازی تجاری. جریان اطلاعات در میان عملیات تجاری به شیوه ای مدلسازی می شود که پاسخگوی پرسش های زیر باشد، کدام اطلاعات، فرآیند تجاری را به پیش می برد، چه اطلاعاتی تولید می شود، چه کسی آنها را تولید می کند؟ اطلاعات کجا می روند؟ چه کسی آنهارا پردازش می کند؟

مدلسازی داده ای. جریان اطلاعاتی که به عنوان بخشی از فاز مدلسازی تجارتی تعریف شد، به صورت مجموعه ای از اشیا پالایش می شود که برای پشتیبانی تجارت مورد نیازند، ویژگی های(صفات) هر شیءِ مشخص شده، رابطه میان این اجزا تعیین می شود.

مدل سازی فرآیند. اشیای داده ای تعیین شده در فاز مدل سازی داده ای تبدیل می شوند تا جریان اطلاعات موردنیاز برای پیاده سازی یک عمل تجارتی به دست آید، توصیفات پردازشی برای افزودن، اصلاح کردن، حذف کردن یا بازیابی کردن یک شی داده ای ایجاد می شوند.

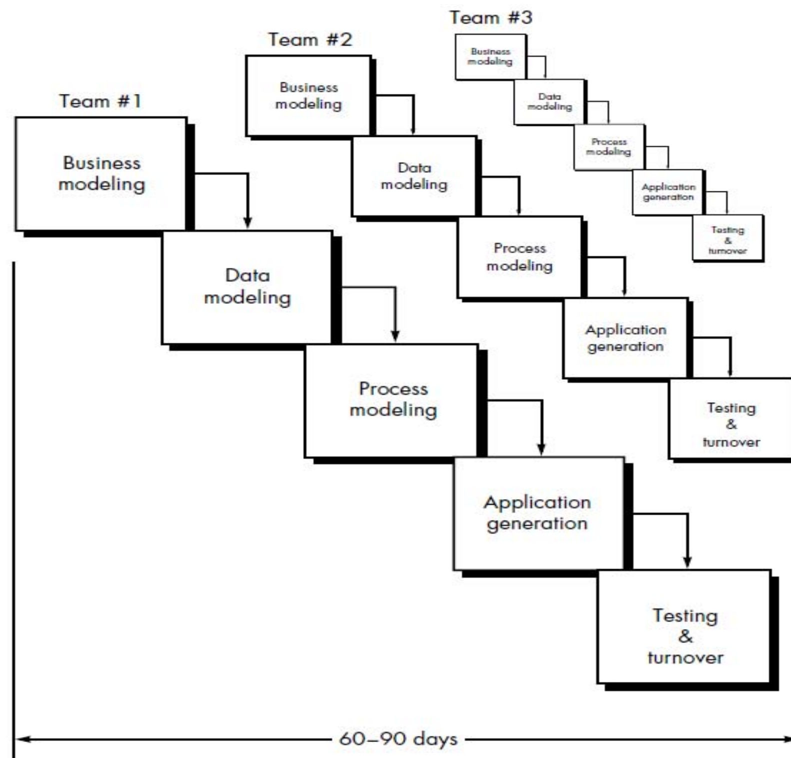
تولید برنامه کاربردی. RAD از تکنیک های نسل چهارم استفاده می کند فرآیند RAD به جای تولید نرم افزار با استفاده از زبان های برنامه نویسی، از مولفه های برنامه موجود (در صورت امکان) استفاده کرده یا مولفه های قابل استفاده مجدد را (در صورت نیاز) ایجاد می کند در همه موارد از ابزارهای خودکار برای تسهیل در امر ساخت نرم افزار استفاده می شود.

آزمایش و توان عملیاتی. چون فرآیند RAD بر استفاده مجدد تاکید دارد بسیاری از مولفه های برنامه قبلا مورد آزمایش قرار گرفته اند. این امر باعث کاهش زمان کلی آزمایش می شود ولی مولفه های جدید باید آزمایش شوند و با همه واسطها تمرین کامل شود.

مدل فرآیند RAD در شکل ۶-۲ نشان داده شده است واضح است که مرزهای زمانی تحمیل شده بر یک پروژه RAD دامنه ای میزان پذیر را طلب می کند اگر یک کاربرد تجارتی را بتوان به شیوه ای تقسیم بندی کرد که هر وظیفه اصلی در کمتر از سه ماه(با استفاده از روش شرح داده شده در بالا) کامل شود؛ این کاربرد کاندیدای خوبی برای روش RAD خواهد بود هر وظیفه اصلی را می توان بر عهده یک تیم RAD جداگانه گذاشت و سپس همه وظایف را یکپارچه نمود.

روش RAD نیز همانند همه مدل های فرآیند، نتایج سوءِ دارد:

- برای پروژه های بزرگ ولی میزان پذیر، RAD برای ایجاد تعداد مناسبی از تیم های RAD به منابع انسانی کافی نیاز دارد.
- RAD نیاز به سازندگان و مشتریانی دارد که نسبت به فعالیت های سریع مورد نیاز برای کامل کردن سیستمی در یک چارچوب زمانی فشرده تر معتقد هستند، اگر هر یک از طرفین فاقد تعهد لازم باشد، پروژه RAD به شکست خواهد انجامید.
- همه انواع کاربردها برای RAD مناسب نیستند اگر سیستمی را نتوان به طور مناسب تقسیم بندی کرد، ساخت مولفه های لازم برای RAD مشکل آفرین خواهد شد، اگر کارآیی بالایی مد نظر باشد و نیل به این کارآیی از تنظیم واسطهایی به مولفه های سیستم میسر باشد، روش RAD ممکن است جواب ندهد.



شکل ۶-۲ مدل RAD

- وقتی احتمال بروز خطرات فنی بالا باشد، RAD مناسب نیست این وضعیت هنگامی رخ می دهد که یک کاربرد جدید، از یک فناوری جدید، استفاده گسترده ای به عمل می آورد یا وقتی که نرم افزار جدید به درجه بالایی از قابلیت همکاری با برنامه های کامپیوتری موجود، نیاز دارد.

۲-۷ مدل های تکاملی فرآیند نرم افزار

رفته رفته این احساس تقویت می شود که نرم افزارها همانند همه سیستم های پیچیده دیگر، در اثر مرور زمان تکامل می یابند خواسته های تجارتی محصول غالبا به موازات توسعه، تغییر می یابند و منجر به ساخت محصول نهایی غیر واقعی می شوند، مهلت های زمانی محدود بازار، کامل کردن یک محصول نرم افزار مفهومی را غیر ممکن می سازند ولی یک نسخه محدود را باید وارد بازار کرد تا فشارهای رقابتی یا کاری را مرتفع سازد، مجموعه ای از خواسته های اصلی و محوری سیستم یا محصول به خوبی درک می شود، ولی جزئیات محصول یا سیستم هنوز باید مشخص شود در این وضعیت ها یا اوضاع مشابه، مهندسان نرم افزار به مدل فرآیندی نیاز دارند که به طور مشخص برای محصول طراحی شده است و با گذشت زمان تکامل می یابند.

مدل ترتیبی خطی (بخش ٤-٢) برای توسعه آسان طراحی شده است در اصل در این روش آبشاری فرض می شود که یک سیستم کامل پس از طی شدن یک ترتیب خطی، آماده تحویل است. مدل ساخت نمونه اولیه (بخش ۵-۲) برای کمک به مشتری (یا سازنده) در شناخت خواسته ها طراحی می شود، به طور کلی این مدل برای تحویل یک سیستم آماده طراحی نشده است ماهیت تکاملی نرم افزار در هیچ یک از این الگوهای کلاسیک مهندسی نرم افزار در نظر گرفته نشده است.

مدل های تکاملی، تکراری هستند، این مدل ها به شیوه ای طراحی می شوند که مهندس نرم افزار را قادر می سازند تا نسخه هایی از نرم افزار را توسعه دهد که هر یک از قبلی کامل تر است.

۱-۷-۲ مدل گام به گام

مدل گام به گام، عناصر مدل ترتیبی خطی را (با اجرای مکرر آنها) با فلسفه تکراری مدل ساخت نمونه اولیه تلفیق می کند که با رجوع با شکل ۷-۲ مشاهده می شود که مدل گام به گام، مراحل ترتیبی خطی را به شیوه ای باور نکردنی یا پیشرفت زمانی تقویم اجرا می کند. هر ترتیب خطی یک قطعه قابل تحویلی از نرم افزار (گام) را ارائه می دهد. برای مثال، نرم افزار واژه پردازی که با استفاده از الگوی گام به گام توسعه یافته است، ممکن است اعمالی از قبیل مدیریت فایل، تولید و ویرایش مستندات را در گام اول، قابلیت های پیچیده تر ویرایشی و تولید مستندات در گام دوم، چک کردن املاءِ و دستور در گام سوم، و قابلیت های پیشرفته صفحه بندی را در گام چهارم تحویل دهد باید توجه داشت که جریان فرآیند برای هر گام می تواند الگوی ساخت نمونه اولیه را در خود داشته باشد.

هنگامی که از یک مدل گام به گام استفاده شود گام نخست غالبا محصول هسته ای است یعنی به خواسته های پایه می پردازد، ولی بسیاری از ویژگی های مکمل (که برخی معلوم و برخی نا معلوم هستند) تحویل داده نمی شوند. محصول هسته ای توسط مشتری مورد استفاده (یا بازبینی مفصل) قرار می گیرد در نتیجه استفاده و یا ارزیابی، طرحی برای گام بعدی توسعه می یابد، این طرح حاوی اصلاحاتی است که نیازهای مشتری و تحویل قابلیت های و ویژگی های اضافی را بهبود می بخشد، این فرآیند به دنبال تحویل هر قطعه تکرار می شود تا این که محصول کامل تولید شود.

مدل فرآیند گام به گام، همانند مدل ساخت نمونه اولیه(بخش ۵-۲) و روش های تکاملی دیگر، ماهیتی تکراری دارد، ولی برخلاف مدل ساخت نمونه اولیه، مدل گام به گام بر تحویل قطعه ای در هر گام تاکید می ورزد. قطعات اولیه، نسخه های «دست و پا شکسته ای» از محصول نهایی هستند ولی قابلیت ارائه خدمات به کاربر را داشته به عنوان محیطی برای ارزیابی توسط کاربر نیز عمل می کنند.

توسعه گام به گام هنگامی مفید واقع می شود که تعداد کارمندان لازم برای تکمیل پیاده سازی پروژه در مهلت کاری مقرر، در دسترس نباشد گام های اولیه را با تعداد کمتری از افراد می توان پیاده سازی نمود.

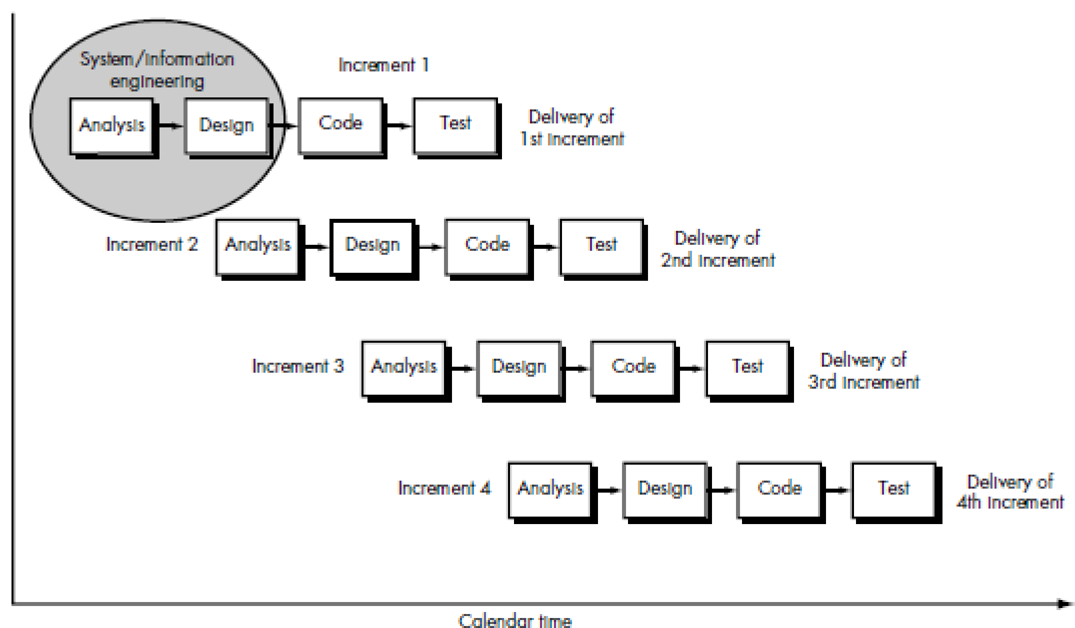


FIGURE 2.7 The incremental model

شکل ۷-۲. مدل گام به گام

۲-۷-۲ مدل مارپیچی یا حلزونی

مدل مارپیچی که نخستین بار بوهم آن را پیشنهاد کرد یک مدل فرآیند نرم افزاری تکاملی است که ماهیت تکراری مدل ساخت نمونه اولیه را با جنبه های کنترلی و سیستماتیک مدل ترتیبی خطی تلفیق می کند. این مدل پتانسیل لازم برای بسط سریع نسخه های تکاملی نرم افزار را دارا می باشد با استفاده از مدل مارپیچی، نرم افزار به صورت یک سری نگارش های تکاملی توسعه می یابد طی نخستین تکرارها، نگارش تکاملی ممکن است یک مدل کاغذی یا یک نمونه اولیه باشد، طی تکرارهای بعدی، هر بار نسخه کاملتری از سیستم، مهندسی شده تولید می شود.

مدل مارپیچی به چند فعالیت چارچوبی تقسیم می شود که نواحی کاری نیز نامیده می شوند به طور متداول بین ۳ تا ۶ ناحیه کاری وجود دارد. شکل ۸-۲ یک مدل مارپیچی را نشان می دهد که حاوی شش ناحیه کاری است.

- **ارتباط با مشتری.** وظایف مربوط به برقراری ارتباط موثر میان سازنده و مشتری.
- **طرح ریزی.** وظایف لازم برای تعیین منابع، خطوط زمانی و دیگر اطلاعات مرتبط با پروژه.
- **تحلیل ریسک.** وظایف مورد نیاز برای ارزیابی خطرات مدیریتی و فنی
- **مهندسی.** وظایف لازم برای ساخت یک یا چند شکل نمایشی از برنامه کاربردی.
- **ساخت و ارائه.** وظایف مورد نیاز برای ساخت، آزمایش، نصب و ارائه پشتیبانی به کاربر (مانند مستندسازی و آموزش).
- **ارزیابی مشتری.** وظایف لازم برای به دست آوردن باز خورد مشتریان براساس ارزیابی شکل های نمایشی نرم افزار که طی مرحله مهندسی ایجاد و طی مرحله نصب، پیاده سازی می شوند.

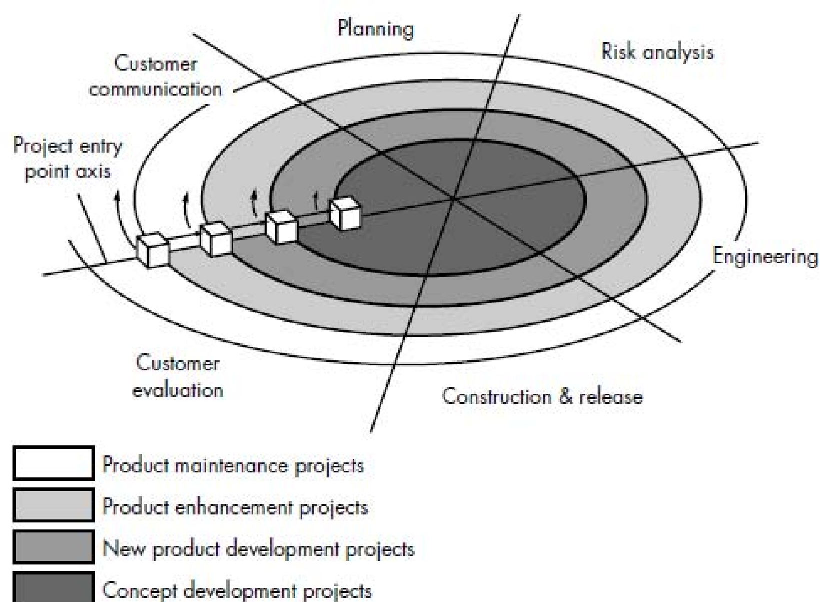
هر یک از نواحی فوق، حاوی مجموعه ای از ظایف کاری است که مجموعه وظایف خوانده می شود این مجموعه وظایف با ویژگی های پروژه همخوانی دارد، برای پروژه های کوچک، تعدادی وظایف کاری و رسمیت آنها کم است برای پروژه های بزرگتر و حیاتی، هر ناحیه کاری حاوی وظایف کاری بیشتری است که برای دستیابی به سطح بالاتری از رسمیت تعریف می شوند در همه موارد، فعالیت پوششی (مانند مدیریت پیکربندی و تضمین کیفیت نرم افزار) که در بخش ۲-۲ ذکر شد، قابل اجرا هستند.

با شروع شدن این فرآیند تکاملی، تیم مهندسی نرم افزار در جهت حرکت عقربه های ساعت و با شروع از مرکز مارپیچ، آن را طی می کند نخستین مدار مارپیچ ممکن است منجر به ایجاد مشخصه ای از محصول می گردد، عبورهای بعدی در اطراف مارپیچ برای ایجاد یک نمونه اولیه و سپس نسخه های پیچیده تری از نرم افزار به کار می رود هر بار گذر از ناحیه طرح ریزی، منجر به تنظیم دوباره طرح پروژه می شود هزینه و زمانبندی براساس بازخورد حاصل از ارزیابی مشتری تنظیم می شود، به علاوه، مدیر پروژه تعداد تکراری های مورد نیاز برای کامل شدن نرم افزار را تنظیم می کند.

برخلاف مولد های فرآیند کلاسیک که با تحویل نرم افزار پایان می یابند، مدل مارپیچی را می توان طوری تطبیق داد که در سرتاسر عمر نرم افزار کامپیوتری قابل به کارگیری باشد یک راه دیگر برای نگریستن به مدل مارپیچی، بررسی محور نقطه های ورود به پروژه است که آن نیز در شکل ۸-۲ نشان داده شده است هر یک از مکعب های قرار داده شده در راستای محور را می توان برای نمایش نقطه آغاز انواع متفاوتی از پروژه ها به کاربرد. یک پروژه بسط مفهوم در مرکز مارپیچ آغاز می شود و آنقدر ادامه می یابد (چندین تکرار را در راستای مسیر مارپیچی رخ می دهد که ناحیه هاشور زده در مرکز را مرزبندی می کند) تا بسط مفهوم کامل شود، اگر قرار باشد این مفهوم در یک محصول واقعی تجلی پیدا کند، فرآیند از طریق مکعب بعدی (نقطه بعدی ورود به پروژه در بسط محصول جدید)

مدل مارپیچی یک روش واقع گرا برای بسط نرم افزارها و سیستم هایی در مقیاس بزرگ است از آنجا که نرم افزار به موازات پیشرفت فرآیند، تکامل می یابد، سازنده و مشتری در هر سطح تکامل، ریسک ها را بهتر درک کرده و به آن واکنش نشان می دهند مدل مارپیچی از ساخت نمونه اولیه به عنوان راهکاری برای کاهش ریسک استفاده می کند ولی مهمتر آنکه سازنده را قادر می سازد تا روش ساخت نمونه اولیه را در هر مرحله از تکامل محصول به کار بندد، این مدل، همان روش مرحله ای پیشنهاد شده توسط چرخه حیات کلاسیک را حفظ می کند، ولی آن را با یک چارچوب تکراری همراه می کند که جهان واقعی را واقعی تر منعکس می کند که مدل مارپیچی، در نظر گرفتن ریسک های فنی در همه مراحل، ضروری است و اگر به طور مناسب به کار برده شود، باید ریسک را پیش از آنکه مشکل آفرین شوند، کاهش دهد.

ولی همانند الگوهای دیگر، این مدل نیز علاج همه دردها نیست، ممکن است به سختی بتوان مشتری را قانع کرد (به ویژه در شرایط قرارداد) که روش تکاملی قابل کنترل است این مدل، مهارت ارزیابی خطر فراوانی را طلب می کند، و برای موفقیت بر همین مهارت متکی است، اگر یک خطر عمده کشف و اداره نشود، بدون شک مشکلاتی به بار خواهد آمد. سرانجام این که مدل فوق به اندازه مدل های ترتیبی خطی و ساخت نمونه اولیه به کار گرفته نشده است و سال ها زمان لازم است تا بازدهی این الگوی جدید و مهم با قطعیت مطلق تعیین گردند.



شکل ۸-۲ مدل مارپیچی معمولی

۳-۷-۲ مدل مارپیچی WINWIN

در مدل مارپیچی که در بخش قبل بحث شد، یکی از اعمال چارچوبی، به برقراری ارتباط با مشتری مربوط می شد، هدف این عمل روشن کردن خواسته ها از سوی مشتری است، در حالت ایده آل، سازنده صرفا از مشتری می پرسد که چه چیز مورد نیاز است و مشتری جزئیات لازم برای پیشرفت کار را فراهم می آورد. متاسفانه چنین

در بهترین مباحثات سعی می شود تا یک نتیجه «برد برد» حاص شود. یعنی مشتری با دستیابی به محصول یا سیستمی که واجد اکثر نیازهای اوست برنده می شود و سازنده با کار کردن در چارچوب مهلت و بودجه ای واقعی و قابل حصول، برنده می شود.

مدل مارپیچی WINWIN بوهم مجموعه ای از اعمال مباحثاتی را در آغاز هر دور جدید از مارپیچ تعریف می کند. به جای یک عمل ارتباط با مشتری، اعمال زیر انجام می شود:

۱. شناسایی واگذارنده کلیدی سیستم یا زیر سیستم.

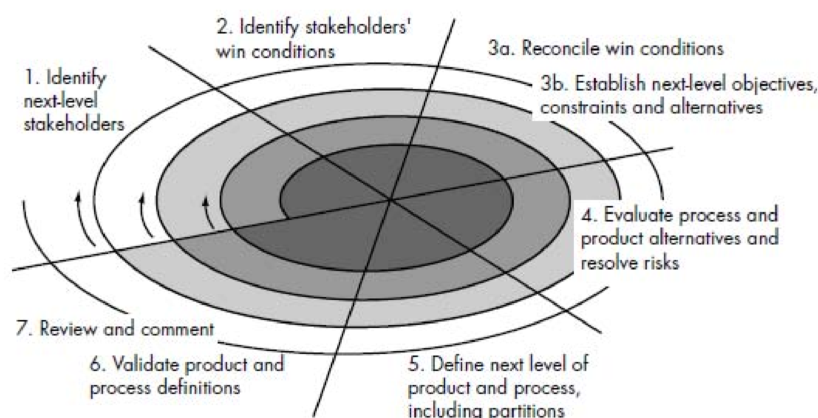
۲. تعیین شرایط برد واگذارنده.

۳. بحث و گفتگو درباره شرایط برد واگذارنده، برای مصالحه و توافق آنها در یک مجموعه شرایط برد برد، برای همه موارد مربوط (از جمله تیم پروژه نرم افزار).

با به پایان بردن موفقیت آمیز این مراحل اولیه، یک نتیجه بردبرد حاصل می شود که ملاک کلیدی برای پیشروی به سمت تعریف نرم افزار و سیستم می شود مدل مارپیچی WINWIN در شکل ۹-۲ نشان داده شده است.

علاوه بر تاکیدی که بر زود هنگام بودن مباحثات می شود، مدل WINWIN سه مرحله فرآیند دارد که نقاط لنگرگاه نام دارند این نقاط به تعیین زمان تکمیل یک چرخه حول مارپیچ کمک کرده نقاط عطفی برای اتخاذ تصمیم، قبل از ادامه پروژه هستند.

در اصل، نقاط لنگرگاه سه دید متفاوت از پیشرفت پروژه را در راستای طی کردن مارپیچ به دست می دهند. نخستین نقطه لنگرگاهی، که اهداف چرخه حیات (LCO) نام دارد، برای فعالیت عمده در مهندسی نرم افزار، مجموعه ای از اهداف را تعیین می کند برای مثال، به عنوان بخشی از LCO، یک مجموعه اهداف با تعریف خواسته های محصول، سیستم سطح بالا همراه می شود دومین نقطه لنگرگاهی، که معماری چرخه حیات (LCA) نام دارد، اهدافی را تعیین می کند که باید به موازات تعریف معماری سیستم و نرم افزار برآورده شوند برای مثال به عنوان بخشی از LCA، تیم پروژه نرم افزار باید تشریح کند که موجودیت مولفه های آماده و قابل استفاده مجدد را ارزیابی کرده، تاثیر آنها را بر تصمیم گیری های معماری مدنظر قرار داده است، قابلیت عملیاتی اولیه (ICO) سومین نقطه لنگرگاهی است و مجموعه ای از اهداف است که عبارتند از آماده سازی نرم افزار جهت نصب، توزیع، آماده سازی سایت پیش از نصب، و کمک به تمام کسانی که نرم افزار را استفاده یا پشتیبانی می کنند.



شکل ۹-۲ مدل مارپیچی WINWIN

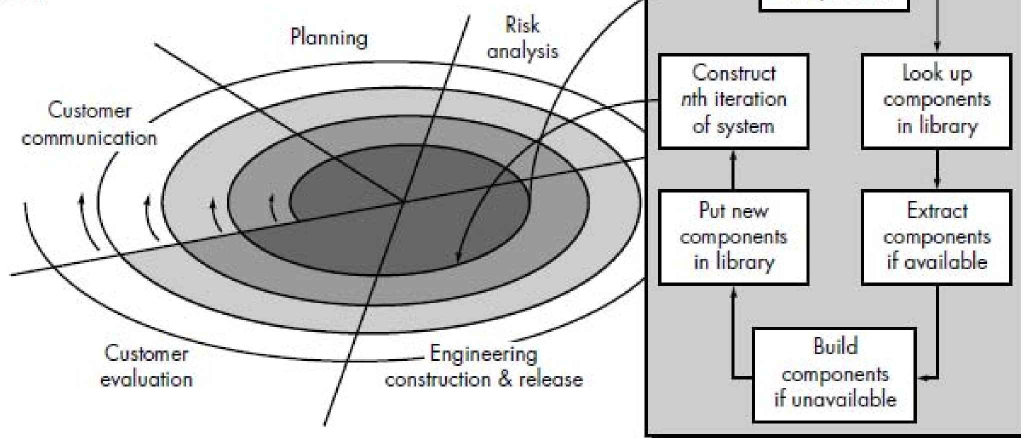
۸-۲ بسط مبتنی بر مولفه ها

فناوری شی گرا چارچوبی فنی برای مدل فرآیند مبتنی بر مولفه جهت مهندسی نرم افزار فراهم می آورد الگوی شی گرا بر ایجاد کلاس هایی تاکید می ورزد که هم داده ها و هم الگوریتم های مورد استفاده برای دستکاری آن داده ها را پنهان سازی می کنند کلاس های شی گرا در صورتی که درست طراحی شوند در کاربردها و معماری های کامپیوتری متفاوت قابل استفاده خواهند بود.

مدل بسط مبتنی بر مولفه ها (CBD) (شکل ۱۱-۲) بسیاری از ویژگی های مدل مارپیچی را در خود دارد، این مدل ماهیتی تکاملی داشته [NIE92] روشی تکراری برای خلق نرم افزار طلب می کند ولی مدل بسط مبتنی بر مولفه ها، برنامه های کاربردی را با استفاده از مولفه های نرم افزاری بسته بندی شده، (که در شکل ۱۱-۲ کلاس نامیده شده اند) می سازد.

فعالیت مهندسی با شناخت کلاس های کاندیدا آغاز می شود این هدف با بررسی داده هایی برآورده می شود که قرار است توسط برنامه کاربردی و الگوریتم مورد استفاده برای دستکاری آن داده ها، دستکاری شوند داده ها و الگوریتم های متناظر در یک کلاس بسته بندی می شوند.

کلاس های ایجاد شده در پروژه های مهندسی نرم افزار قبلی، در یک کتابخانه یا مخزن کلاس ها نگهداری می شوند هنگامی که کلاس های کاندیدا تعیین شدند، کتابخانه کلاس ها مورد جستجو قرار می گیرد تا معلوم شود که آیا این کلاس ها از قبل وجود دارند اگر وجود داشته باشند، از کتابخانه استخراج و دوباره استفاده می شوند اگر کلاس کاندیدایی در کتابخانه نبود، با استفاده از روش های شی گرا ایجاد می شود سپس نخستین تکرار از برنامه کاربردی با استفاده از کلاس های استخراج شده از کتابخانه و کلاس های جدید ساخته می شود تا نیازهای منحصر بفرد آن برنامه کاربردی برآورده شود. سپس جریان فرآیند به مارپیچ باز می گردد و سرانجام در اثنای گذرهای بعدی از میان فعالیت مهندسی دوباره وارد تکرار مونتاژ مولفه ها می شود.



شکل ۱۱-۲ بسط مبتنی بر مولفه ها

توسعه مبتنی بر مولفه ها منجر به استفاده مجدد از نرم افزار می شود و این قابلیت استفاده مجدد برای مهندس نرم افزار چند مزیت عمده به ارمغان می آورد. بنگاه QSM Associates براساس مطالعاتی که روی قابلیت استفاده مجدد انجام داده است چنین گزارش می کند که مونتاژ مولفه های منجر به کاهش ۷۰ درصدی در زمان چرخه توسعه نرم افزار، کاهش ٤٨ درصدی در هرینه پروژه و ضریب بهره وری ۲۶/۲ در مقابل هنجار صنعتی ۱۶/۹ می شود. گرچه این نتایج تابعی از غنای کتابخانه مولفه ها است، با کمترین تردید می توان گفت که مدل توسعه مبتنی بر مولفه های مزایای چشمگیری برای مهندسان نرم افزار دارد.

فرآیند بسط نرم افزار یکنواخت، نماینده چند مدل بسط مبتنی بر مولفه ها است که در صنعت پیشنهاد شده اند زبان مدل سازی یکنواخت (UML)، فرآیند یکنواخت مولفه هایی را که برای ساخت سیستم به کار می روند و واسط هایی را که برای متصل ساختن مولفه ها به یک دیگر مورد استفاده قرار می گیرند، معین می سازد. فرآیند یکنواخت با استفاده از تلفیق توسعه تکراری و گام به گام، عملکرد سیستم را با اعمال یک روش مبتنی سناریو (از دیدگاه مشتری) تعریف می کند سپس عملکرد را با یک چارچوب معماری که شکل نهایی نرم افزار تعیین می کند تلفیق می نماید.

۹-۲ مدل روش های رسمی

مدل روش های رسمی شامل مجموعه ای از فعالیت ها است که به مشخص کردن ریاضی و رسمی نرم افزار کامپیوتری منجر می شود روش های قرار دادی، مهندس نرم افزار را قادر می سازد تا با اعمال یک نظم ریاضی شدید، سیستم کامپیوتری را مشخص کند، بسط دهد و مورد تصدیق قرار دهد شکل دیگری از این روش، که مهندسی نرم فزار اتاق تمیز نامیده می شود. در حال حاضر توسط برخی سازمان های نرم افزار سازی به کار می رود.

هنگامی که روش های قراردادی در اثنای بسط نرم افزار به کار برده می شوند، راهکاری برای حذف بسیاری از مشکلات فراهم می آوردند که غلبه بر آنها با استفاده از الگوهای مهندسی دیگر، دشوار است ابهام، ناقص بودن و ناسازگاری را می توان راحت تر کشف و تصحیح کرد، نه از طریق بازبینی خاص بلکه از طریق به کارگیری تحلیل ریاضی. هنگامی که روش های رسمی در اثنای طراحی به کار برده می شوند، به عنوان مبنایی برای معتبر سازی برنامه عمل کرده از اینرو مهندس نرم افزار را قادر به کشف و تصحیح خطاهایی می سازد که ممکن بود و در غیر این صورت ناپیدا بمانند.

مدل روش های رسمی گرچه چندان عمومیت نخواهد یافت. نویدبخش نرم افزاری عاری از نقص است با این حال ملاحظات مربوط به قابلیت اجرای آن در محیط های تجارتی چنین اعلام شده است.

۱. بسط مدل های رسمی در حال حاضر بسیار وقت گیر و پر هزینه است.

۲. از آنجا که تعداد معدودی از نرم افزار سازان زمینه لازم برای اجرای روش های رسمی هستند آموزش گسترده ای مورد نیاز است.

۳. استفاده از مدل ها به عنوان راهکار ارتباطی با مشتریانی که دید فنی ندارند دشوار است.

نظر به این ملاحظات، روش های رسمی احتمالا در میان نرم افزار نویسانی هوادار پیدا می کند که باید نرم افزارهای بحرانی – امنیتی (مثلا نرم افزارهای دستگاه های پزشکی و هوا فضا) بسازند و یا در میان آنهایی که در صورت بروز خطا در نرم افزار دستخوش زیان های اقتصادی کلان می شوند.

۱۰-۲ تکنیک های نسل چهارم.

تکنیک های نسل چهارم (4GT) شامل آرایه وسیعی از ابزارهای نرم افزاری است که در یک چیز مشترک هستند هر یک از این ابزارها مهندس نرم افزار را قادر می سازند تا یک ویژگی از نرم افزار را در سطحی بالا مشخص کند سپس این ابزارها به طور خودکار کد منبع را براساس مشخصه سازنده تولید می کنند این بحث کوچک وجود دارد که هر چه سطح مشخص سازی نرم افزار بالاتر باشد، برنامه را سریع تر می توان ساخت الگوی 4GT برای مهندسی نرم افزار، بر توانایی مشخص کردن نرم افزار با استفاده از اشکال زبانی یا نشانه گذاری های گرافیکی تاکید دارد تا مسئله مورد نظر را به صورتی حل کند که برای مشتری قابل درک باشد.

در حال حاضر، یک محیط نرم افزاری که الگوی 4GT را پشتیبانی کند شامل برخی ابزارهای زیر با همه آنها می شود: زبان های غیر رویه ای برای درخواست بانک های اطلاعاتی، تولید گزارش، دستکاری داده ها، تعریف صفحه نمایش و تعامل با آن، تولید کد، قابلیت گرافیک سطح بالا، قابلیت صفحه گسترده و تولید خودکار HTML و زبان های مشابه دیگر مورد استفاده برای ایجاد سایت های وب با به کارگیری ابزارهای نرم افزاری پیشرفته، در آغاز، فقط برای دامنه های کاربردی بسیار مشخص در دسترس بود، ولی امروزه محیط های 4GT طوری توسعه یافته اند که از عهده اکثر گروه های کاربردی نرم افزار بر می آیند.

4GL نیز همانند الگوهای دیگر با مرحله جمع آوری خواسته ها آغاز می شود به طور ایده آل، مشتری خواسته ها را برمی شمرد و این خواسته ها مستقیما به یک نمونه اولیه کاری منتقل می شوند، ولی این کافی نیست ممکن است مشتری از نیازهای خود مطمئن نباشد ممکن است حقایق دانسته شده را با ابهام مشخص کرده باشد و ممکن است نتواند یا نخواهد اطلاعات را به شیوه ای مشخص کند که برای 4GT قابل مصرف باشد به همین دلیل گفتگوی میان سازنده و مشتری که برای مدل های فرآیند دیگر شرح داده شد، همچنان یک بخش ضروری در روش 4GL است.

برای کاربردهای کوچک شاید بتوان با استفاده از یک زبان نسل چهارم غیر رویه ای (4GL) یا مدلی مرکب از شبکه آیکن های گرافیکی، مرحله جمع آوری خواسته ها را پیاده سازی نمود ولی برای کارهای بزرگ تر، لازم است تا یک راهبرد طراحی برای سیستم توسعه یابد، حتی اگر قرار باشد از یک 4GL استفاده شود استفاده از 4GL بدون طراحی (برای پروژه های بزرگ) باعث همان مشکلاتی (کیفیت پایین؛ قابلیت نگهداری ضعیف، پذیرش ضعیف از طرف مشتری) می شود که هنگام توسعه نرم افزار با روش های مرسوم، با آنها مواجه می شویم.

پیاده سازی با استفاده از یک 4GL، سازنده نرم افزار را قادر می سازد تا نتایج مطلوب را به شیوه ای ارائه می دهد که منجر به تولید خودکار کد برای ایجاد آن نتایج شود واضح است که ساختمان داده ای با اطلاعات مفید باید موجود باشد و به راحتی در دسترس 4GL باشد.

سازنده برای تبدیل پیاده سازی 4GT به محصول باید آزمایش کامل به عمل آورد، مستندات با معنی توسعه دهد و همه فعالیت های یکپارچه سازی دیگری را که در الگوهای مهندسی نرم افزار دیگر مورد نیاز هستند، اجرا نماید، به علاوه نرم افزار توسعه یافته به شیوه 4GT باید به صورتی ساخته شود که نگهداری محصول به طرزی کارآمد میسر باشد.

همانند کلیه الگوهای مهندسی نرم افزار، مدل 4GT نیز دارای مزایای و معایبی است هواداران این مدل ادعا می کنند زمان توسعه نرم افزار کاهش زیادی می یابد و تا حد زیادی بهره وری افرادی که نرم افزار را می سازند، بالاتر می رود و مخالفان آن ادعا می کنند که استفاده از ابزارهای کنونی 4GT چندان هم آسانتر از به کارگیری زبان های برنامه نویسی نیست، کد منبع تولید شده «کارآمد» نیست و قابلیت نگهداری سیستم های نرم افزاری با استفاده از 4GT توسعه یافته اند مورد تردید است.

ادعای هر دو طرف تا حدی درست است و می توان وضعیت جاری 4GT را چنین خلاصه کرد:

۱. استفاده از 4GT، روشی عملی برای اکثر زمینه های کاربردی متفاوت است 4GT به همراه ابزارهای مهندسی نرم افزار به کمک کامپیوتر (CASE) و مولدهای کد، راه حلی قانع کننده برای اکثر مشکلات نرم افزار را ارائه می دهد.

۲. داده های جمع آوری شده از شرکت هایی که از 4GT استفاده می کنند، نشان می دهد که زمان لازم برای تولید نرم افزار تا حد زیادی برای کاربردهای کوچک و متوسط کاهش می یابد و نیز میزان تحلیل و طراحی لازم کاربردهای کوچک کاهش می یابد.

۳. استفاده از 4GT برای امور توسعه نرم افزارهای بزرگ، به همان اندازه تحلیل، طراحی و آزمایش (فعالیت های مهندسی نرم افزار) و بلکه بیشتر نیاز دارد در وقت صرفه جویی شود، و می توان از طریق حذف کد نویسی به این هدف دست پیدا کرد.

فصل ۳ : مفاهیم مدیریت پروژه

مدیریت پروژه چیست؟ گر چه بسیاری از ما (لحظات تاریک زندگی خود) به مدیریت، همانند دیلبرت می نگریم، هنگام ساخت محصولات و سیستم های کامپیوتری ضروری به نظر می رسد مدیریت پروژه شامل طرح ریزی نظارت و کنترل افراد، فرآیند و رویدادهایی می شود که به موازات تکامل نرم افزار از مفهوم مقدماتی تا پیاده سازی عملی رخ می دهند.

چه می کند؟ هر کسی تا حدی مدیریت می کند ولی حوزه فعالیت های مدیریتی بسته به شخص تغییر می کند مهندس نرم افزار فعالیت های روزانه و وظایف فنی کنترل، نظارت و طرح ریزی خود را مدیریت می کند مدیران پروژه، کار تیمی از مهندسان نرم افزار را طرح ریزی، نظارت و کنترل می کنند. مدیران ارشد ارتباط میان نرم افزار نویسان حرفه ای و بازار کار را هماهنگ می کنند.

چرا اهمیت دارد؟ ساخت نرم افزارهای کامپیوتری وظیفه ای پیچیده است، به ویژه اگر افراد زیادی در آن شرکت داشته باشند و مدت زمان کار نسبتا طولانی باشد از همین رو است که پروژه ها باید مدیریت شوند.

مراحل کار کدام است؟ در چهار مورد- افراد، محصول، فرآیند، پروژه- افراد را باید طوری سازماندهی کرد که نرم افزاری موثر بسازند برقراری ارتباط با مشتری باید طوری صورت پذیرد که نیازمندی های و حوزه محصول درک شود فرآیندی باید انتخاب شود که برای افراد و محصول مورد بحث مناسب باشد پروژه باید با برآورد کار و زمان لازم جهت انجام وظایف طرح ریزی شود، تعیین محصولات کار، برقراری نقاط کنترل کیفیت و وضع راهکارهایی جهت نظارت و کنترل کارهای تعیین شده توسط طرح.

محصول کار چیست؟ در نتیجه فعالیت های مدیریتی، یک طرح پروژه ایجاد می گردد. این طرح، فرآیند و وظایفی را که باید به انجام برسند، افرادی را که کار را انجام می دهند و راهکارهای ارزیابی خطرات، کنترل تغییر و ارزیابی کیفیت را تعیین می کند.

چگونه مطمئن شوم که درست از عهده امور برآمده ام؟ هرگز به طور کامل اطمینان حاصل نخواهید کرد که طرح پروژه شما درست است تا این که محصولی با کیفیت بالا را سر وقت و با بودجه تعیین شده تحویل دهید ولی مدیر پروژه هنگامی کار خود را درست انجام می دهد که افراد خود را به همکاری در قالب تیمی کارآمد تشویق کند و آنها توجه خود را به نیازهای مشتری و کیفیت محصول معطوف کنند.

۳-۲-۱ بازیگران

فرآیند نرم افزار (و هر پروژه نرم افزاری) شامل بازیگرانی می شود که در یکی از پنج گروه زیر قرار می گیرند.

۱. **مدیران ارشد** که مسائل بازار کار را مشخص می کنند و این مسائل غالبا تاثیر چشمگیری بر پروژه دارند.
۲. **مدیران پروژه** (فنی) که باید برای سازندگان نرم افزار برنامه ریزی کنند، در آنها انگیزه ایجاد کنند، آنها را سازماندهی و کنترل کنند.
۳. **سازندگان** که حائز مهارت های فنی مورد نیاز برای مهندسی یک محصول یا کاربرد هستند.
٤. **مشتریانی** که نیازمندی های نرم افزار و دیگر واگذارندگان موجود در نتیجه کار را مشخص می کنند.
٥. **کاربران نهایی** که در زمان روانه شدن نرم افزار به بازار کار با آن به تعامل می پردازند.

برای آنکه تیم موثر واقع شود، باید به شیوه ای سازماندهی شود که مهارت ها و توانایی های تک تک افراد به حداکثر مقدار برسد.

۳-۲-۲ رهبران تیم

مدیریت پروژه یک فعالیت وابسته به افراد است و از همین رو، شرکای رقیب غالبا رهبران تیم ضعیفی محسوب می شوند، نکته فقط در این است که ترکیب درستی از مهارت های افراد را در دست ندارند، با این حال اجمون می گوید. بدبختانه و چنانچه به کرات دیده می شود افراد وارد یک حیطه مدیریت می شوند و تصادفا مدیر پروژه می شوند.

جری واینبرگ [WEI86] در یک کتاب عالی در خصوص رهبری پروژه های فنی، مدل MOI را برای رهبری پیشنهاد می کند:

انگیزش. توانایی تشویق افراد به تولید با بهترین توانایی آنها.

سازماندهی. توانایی استفاده از فرآیندهای موجود (یا ابداع فرآیندهای جدید) که انتقال مفهوم اولیه به یک محصول نهایی را میسر سازد.

ایده ها و نوآوری ها. توانایی تشویق افراد به خلاقیت و ایجاد احساس خلاقیت حتی زمانی که باید در قید مرزهای تعیین شده برای یک محصول یا کاربرد خاص کار کنند، واینبرگ پیشنهاد می کند که رهبران پروژه های موفق، شیوه مدیریتی حل مسئله را به کار می بندند، یعنی مدیر پروژه نرم افزاری باید مسئله ای را که قرار است حل شود، درک کند جریان ایده ها را مدیریت کند و در عین حال به اطلاع تمامی افراد گروه برساند (با گفتار و مهمتر از آن با کردار خود) که کیفیت بیشترین اهمیت را دارد و نمی توان از آن عدول کرد.

یک دیدگاه دیگر از ویژگی های مدیریت پروژه موثر، بر چهار نکته کلیدی زیر تاکید دارد:

حل مسئله. یک مدیر پروژه موثر می تواند مشکلات سازمانی و فنی موجود را تشخیص دهد به طور سیستماتیک راهکاری برای آنها ارائه دهد یا به طرزی مناسب در سازندگان دیگر ایجاد انگیزه کند تا راهکاری ارائه دهند، تجربیات اندوخته شده از پروژه های قبلی را در شرایط جدید به کار بندد و اگر کوشش های اولیه برای حل مسئله به ثمر نرسید، انعطاف پذیری لازم برای تغییر جهت گیری را داشته باشد.

هویت مدیریتی. یک مدیر پروژه خوب باید مسئولیت پروژه را بر عهده بگیرد او باید ضمانت لازم برای آزاد گذاشتن افراد فنی در دنبال کردن غرایزشان را داشته باشد.

نیل به اهداف. مدیر برای بهینه کردن بهره وری تیم پروژه باید به موفقیت ها پاداش دهد و از طریق اعمال خود نشان می دهد که ریسک کردن تحت کنترل منجر به تنبیه نخواهد شد.

تاثیر و ساخت تیم. مدیر کارآمد باید قادر به درک افراد باشد، او باید بتواند سیگنال های کلامی و غیر کلامی را بفهمد. و به نیازهای افراد که این سیگنال ها را ارسال می کنند، واکنش نشان دهد مدیر باید در شرایط فشار کاری بالا نیز کنترل را در دست داشته باشد.

۳-۲-۳ تیم نرم افزار

تقریبا به همان تعداد که سازمان های نرم افزارسازی وجود دارد، ساختارهای سازماندهی افراد نیز وجود دارد بهتر یا بدتر ساختارهای سازماندهی را به آسانی نمی توان اصلاح کرد امور مربوط به پیامدهای عملی و سیاسی

برای به کارگماردن منابع انسانی در پروژه ای که نیاز به کار n نفر به مدت k سال دارد، انتخاب های زیر وجود دارد :

۱. n نفر مکلف به انجام m وظیفه کاری می شوند، کار ترکیبی نسبتا کمی انجام می شود مسئولیت مدیر نرم افزاری که ممکن است شش پروژه، دیگر نیز داشته باشد هماهنگ سازی است.

۲. n نفر مکلف به انجام m وظیفه کاری می شوند (m<n) به طوری که تیم های غیر رسمی تشکیل می شوند. ممکن است یک رهبر خاص برای تیم تعیین شود مسئولیت مدیر نرم افزار هماهنگ سازی است.

۳. n نفر در قالب t تیم سازماندهی می شوند؛ هر تیم مکلف به انجام یک یا چند وظیفه کاری می شود، هر تیم دارای ساختار مشخص است که برای همه تیم های در حال کار روی پروژه تعیین شده است هماهنگ سازی هم توسط تیم و هم مدیر پروژه نرم افزاری صورت می پذیرد گر چه امکان اقامه استدلالاتی بر علیه و بر له هر یک از روش های بالا وجود دارد، شواهد فزاینده ای وجود دارد که نشان می دهد سازماندهی رسمی تیم ها (انتخاب ۳) بیشترین بهره وری را دارد.

ساختار بهترین تیم به شیوه مدیریت سازمان، به تعداد افراد تشکیل دهنده تیم و سطوح مهارتی آنان و میزان کلی دشواری مسئله بستگی دارد مانتی سه نوع سازماندهی کلی برای یک تیم پیشنهاد می کند.

تمرکززدایی دموکراتیک (DD). تیم مهندسی نرم افزار فاقد رهبری دائمی است، در عوض هماهنگ کنندگان وظایف برای مدتی کوتاه نصب شده سپس جای خود را به کسان دیگری می دهند که ممکن است وظایف متفاوتی را هماهنگ کنند تصمیم گیری درباره مشکلات و روش کار از طریق اجماع گروه صورت می پذیرد ارتباط میان اعضای تیم، افقی است.

تمرکززدایی کنترل شده (CD). تیم مهندسی نرم افزار دارای رهبری مشخص است که وظایف معینی را هماهنگ می کند و رهبران ثانویه ای مسئول وظایف فرعی هستند حل مشکلات کماکان از فعالیت های گروهی است ولی پیاده سازی راهکارها توسط رهبر تیم، میان زیرگروه ها تقسیم می شود ارتباط میان زیرگروه ها و افراد افقی است همچنین ارتباطی عمودی میان سلسله مراتب کنترلی وجود دارد.

تمرکز کنترل شده (CC). هماهنگ سازی داخلی تیم و حل مشکلات سطح بالا بر عهده مدیر تیم است ارتباط میان رهبر و اعضا عمودی است.

مانتی هفت عامل پروژه را توصیف می کند که باید هنگام طرح ریزی ساختار تیم های مهندسی نرم افزار در نظر گرفت.

- دشواری مسئله ای که قرار است حل شود،
- اندازه برنامه(های) حاصل برحسب تعداد خطوط کد یا تعداد نقاط تابع.
- زمان کنار هم ماندن اعضای تیم.
- میزان قابلیت پیمانه ای مسئله (تقسیم مسئله به اجزای کوچک تر).
- کیفیت و قابلیت اطمینان مورد نیاز برای سیستمی که قرار است ساخته شود.
- قطعیت تاریخ تحویل.
- میزان ارتباطات مورد نیاز برای پروژه.

چون با ساختار متمرکز وظایف سریعتر انجام می شود، بهترین انتخاب برای مسائل ساده به شمار می رود، تیم های غیر متمرکز راهکارهای بیشتر و بهتری نسبت به افراد می یابند بنابراین هنگامی که چنین تیم هایی روی مسائل دشوار کار می کند احتمال موفقیت بیشتر است چون تیم CD برای حل مسئله متمرکز شده است هر یک از ساخترهای CD یا CC را می توان به طور موفقیت آمیزی در مورد مسائل ساده به کاربرد. ساختار DD برای مسائل دشوار از همه بهتر است.

از آنجا که کارآیی یک تیم با مقدار ارتباطاتی که باید برقرار شود، نسبت عکس دارد پروژه های بسیار بزرگتر در صورتی که بتوان آنها را به زیرگروه های کوچکتر تقسیم کرد، به بهترین وجه با ساختارهای CC یا DD قابل انجام هستند.

مدت زمانی که اعضای تیم با یکدیگر هستند بر رفتار تیم تاثیر می گذارد، معلوم شده است که ساختار تیمی DD منجر به رفتاری عالی و رضایت شغلی می شود و بنابراین خوشا به حال اعضای تیمی که مدت طولانی کنار هم باشند.

بهترین حیطه کاربرد ساختار تیمی DD، مسائلی است که قابلیت تقسیم به اجزای کوچکتر کم است، زیرا در این وضعیت حجم بیشتری از ارتباطات مورد نیاز است هنگامی که قابلیت تقسیم بالایی وجود دارد (و افراد می توانند وظایف خاص خود را انجام دهند) ساختار CC یا CD خوب جواب می دهد.

معلوم شده که تیم های CC و CD کمتر از تیم های DD خطا می کنند، ولی این داده ها ارتباط زیادی با فعالیت های تضمین کیفیتی دارند که توسط تیم انجام می شود تیم های غیر متمرکز عموما به زمان بیشتری برای به پایان رساندن پروژه نیاز دارند و در عین حال، هنگامی که ارتباطات بالایی مورد نیاز باشند از همه بهترند.

کنسانتین چهار الگوی سازمانی برای تیم های مهندسی نرم افزار پیشنهاد می کند:

۱. **در الگوی بسته.** تیم در راستای یک سلسله مراتب سنتی از مسئولیت ها(مشابه CC) سازماندهی می شود چنین تیم هایی هنگام ساخت نرم افزارهایی خوب عمل می کنند که نیاز به کارهای روتین دارد، ولی برای نوآوری و کارهای جدید احتمال موفقیت کمتر است.

۲. **در الگوی تصادفی.** تیم ساختار سستی دارد و به ظرفیت تک تک افراد وابسته است در صورت نیاز به کشفیات فنی یا نوآوری، تیمی که از الگوی تصادفی پیروی می کند، بهترین نتیجه را می دهد ولی در صورتی که کارکردی منظم مورد نیاز باشد چنین تیمی به دردسر می افتد.

۳. **در الگوی باز.** کوشش می شود ساختار تیم به شیوه ای باشد که در عین دستیابی به کنترل های خاص الگوی بسته، به حداکثر نوآوری هایی که در یک الگوی تصادفی مشاهده می شوند، دستیابی شود، کارها از طریق همکاری با ارتباطات سنگین و تصمیم گیری های مبتنی بر اجماع انجام می پذیرد تیمی که ساختار آن از الگوی باز پیروی می کند برای حل مسائل پیچیده بسیار مناسب است ولی ممکن است بازدهی تیم های دیگر را نداشته باشد.

٤. **در الگوی همزمان،** اعضای تیم به شیوه ای قطعه قطعه و سازماندهی می شوند که روی قطعات مسئله کار کنند در حالی که ارتباط فعال میان آن ها بسیار کم باشد.

کنستانتین شکل تغییر یافته ای از تمرکززدایی دموکراتیک را پیشنهاد کرده است که در آن به اعضای تیم استقلال خلاقیت می دهد و روش کار آنها را شاید بتوان به بهترین نحو یا اصطلاح هرج و مرج خلاقانه توصیف کرد گر چه روش آزادانگاری در کار نرم افزار، دارای جذبه است کنترل هرج و مرج و رسیدن به تیمی با کارایی بالا باید هدف اصلی یک سازمان مهندسی نرم افزار باشد برای دستیابی به تیمی با کارآیی بالا:

- اعضای تیم باید به یکدیگر اطمینان داشته باشند.

- توزیع مهارت ها مناسب حال مسئله باشد.
- اگر قرار است یکپارچگی حفظ شود، افراد خود کامه را باید از تیم طرد کرد.

سازماندهی تیم هر چه که باشد هدف هر مدیر پروژه ای کمک به ایجاد تیمی است که از خود یکپارچگی نشان می دهد دومارکوولیستر در کتاب خود، این مسئله را مورد بحث قرار می دهند.

معمولا از واژه تیم در جهان کاری، نسبتا به سستی استفاده می کنیم یعنی هر گروه از افراد را که کاری به آنها تکلیف شده، تیم می نامیم ولی بسیاری از این گروه ها شباهتی به تیم ندارند، هیچ تعریف مشترکی از موقعیت یا روح تیمی قابل تشخیص در آنها نیست آنچه کم است پدیده ای ایست که ما آن را ژل می نامیم.

تیم ژل شده گروهی از افراد است که چنان به هم محکم پیوند خورده اند که کلیت آنها بزرگتر از حاصل جمع تک تک آنهاست

با ژل شدن یک تیم احتمال موفقیت بالا می رود. جلوی تیم را نمی توان گرفت مثل بولدورزی که به سوی موفقیت پیس می رود. نیازی به مدیریت آنها به شیوه های سنتی نیست و قطعا نیازی به انگیزش ندارند آنها تکانه لازم را دارند دمارکوولیستر معتقدند بهره وری و انگیزه اعضای تیم ژل شده به مراتب بیشتر از حد متوسط است آنها هدفی مشترک، فرهنگی مشترک، و در بسیاری موارد، نوعی حس برتری دارند که آنها را منحصر بفرد می سازد. ولی همه تیم ها قابلیت ژل شدن ندارد در واقع، بسیاری تیم ها از پدیده ای رنج می برند که جکمن آن را مسمویت تیم می نامد. او پنج عامل را مشخص می کند که محیط تیم را شدیدا مسموم می کند:

١. یک فضای کاری آشفته که در اعضای تیم انرژی خود را هدر می دهند و توجه خود را معطوف اهداف کاری باید انجام گردد، نمی کنند.

٢. ناراحتی های زیاد ناشی از عوامل فنی، کاری و پرسنلی که باعث اصطکاک میان اعضای تیم می شوند.

٣. رویه های قطعه قطعه شده یا با هماهنگی ضعیف یا مدل فرآیندی که خوب مشخص نشده باشد یا خوب انتخاب نشده باشد و باعث ایجاد مانع شود؛

٤. تعریف ناواضح نقش ها که منجر به فقدان مسئولیت پذیری و انگشت نما شدن می شود.

٥. قرار گرفتن پیوسته و مکرر در معرض شکست که منجر به از دست رفتن اعتماد به نفس و سوءِ رفتار می شود.

جکمن چند پادزهر برای مشکلات ذکر شده در بالا پیشنهاد می کند.

برای پرهیز از محیط کاری آشفته مدیر پروژه باید یقین داشته باشد که تیم به کلیه اطلاعات لازم برای انجام کار دستیابی دارد و اهداف اصلی، هنگامی که تعیین شد، نباید تغییر یابد مگر آنکه واقعا ضروری باشد به علاوه خبرهای بد را نباید مسکوت گذاشت بلکه باید هر چه زودتر به اطلاع تیم رساند(تا زمان برای نشان دادن واکنش به شیوه ای موجه و کنترل شده باقی است).

گرچه ناراحتی علل متعدد دارد، افرادی که در زمینه نرم افزار کار می کنند غالبا هنگامی احساس ناراحتی می کنند که فاقد قدرت لازم برای کنترل اوضاع باشند یک تیم نرم افزار در صورتی می تواند از بروز ناراحتی جلوگیری کند که مسئولیت لازم برای تصمیم گیری را دارا باشد. هر چه کنترل بیشتری روی فرآیند و تصمیم گیری های فنی به تیم اعطا شود، اعضای تیم کمتر احساس ناراحتی می کنند

به دو شیوه می توان از انتخاب فرآیند نرم افزار نامناسب (مثلا وظایف کاری بیهوده و کسل کننده یا محصولات کاری که خوب انتخاب نشده باشند)پرهیز کرد ١) یقین داشتن از این که ویژگی های نرم افزار از قدرت فرآیند انتخاب

مدیر پروژه نرم افزاری که همراه با تیم کار می کند باید به طور واضح وظایف و مسئولیت ها را پیش از آغاز پروژه مشخص کند خود تیم باید راهکاری برای مسئولیت پذیری وضع کند (مرور فنی راهی عالی برای نیل به این هدف است) و یک سری روش های اصلاحی برای خطای افراد تیم مشخص نماید.

هر تیم نرم افزاری شکست های کوچک را تجربه می کند کلید پرهیز از فضای شکست، برقرار کردن تکنیک های تیمی برای بازخورد و حل مسئله است به علاوه هر شکست در کار یکی از اعضای گروه باید به عنوان شکستی برای خود تیم منظور گردد این منجر به روش تیم گرا برای انجام اعمال اصلاحی می شود نه انگشت نما کردن یکی از اعضا و ایجاد عدم اطمینان که در تیم های مسموم به سرعت رشد می کند.

علاوه بر پنج سمی که جکمن توصیف می کند، تیم نرم افزاری غالبا با اختلاف تمایلات انسانی در اعضا نیز مشکل دارد برخی از افراد گوشه گیر بوده برخی با دیگران می جوشند، برخی افراد اطلاعات را با هوشمندی جمع آوری می کنند و مفاهیمی عمیق از یک سری حقایق مجزا به دست می آورند، برخی دیگر، اطلاعات را به طور خطی جمع آوری کرده جزئیات دقیق را از داده های فراهم آمده جمع آوری و سازماندهی می کنند برخی اعضای تیم فقط وقتی در تصمیم گیری راحت هستند که استدلالی منطقی و منظم ارائه شود برخی دیگر که هوشمند هستند، مایلند براساس احساس خود تصمیم گیری کنند برخی خواستار یک برنامه زمانبندی مشروح آکنده از وظایف سازمانی هستند که آنها را قادر می سازد به عنصری از پروژه دست پیدا کنند عده ای دیگر، یک محیط خود به خودی را ترجیح می دهند که در آن بروز مشکلات امری عادی است برخی سخت کار می کنند که کارها را مدت ها قبل از زمان تعیین شده انجام دهند و از فشاری که در روزهای پایانی وجود دارد در امان بمانند، در حالی که عده ای دیگر به سبب کمی فرصت باقی مانده انرژی می گیرند. ذکر این نکته اهمیت دارد که شناخت تفاوت های انسانی نخستین گام در راه ایجاد تیم هایی است که ژل می شوند.

۳-٤ فرآیند

فازهای کلی که فرآیند نرم افزار را مشخص می کنند تعریف توسعه و پشتیبانی در مورد همه نرم افزارهای لازم الاجرا هستند مشکل انتخاب مدل فرآیند برای نرم افزاری است که باید توسط یک تیم پروژه مهندسی شود.

در فصل ۲ گستره وسیعی از الگوهای مهندسی نرم افزار مورد بحث قرار می گیرند.

- مدل ترتیبی خطی
- مدل ساخت نمونه اولیه
- مدل RAD
- مدل گام به گام
- مدل مارپیچی
- مدل بردبرد مارپیچی
- مدل توسعه مبتنی بر مولفه ها
- مدل توسعه همزمان
- مدل روش های رسمی
- مدل تکنیک های نسل چهارم

مدیر پروژه باید تصمیم بگیرد که با در نظر گرفتن موارد زیر کدام مدل از همه مناسب تر است ۱) مشتریانی که محصول را در خواست کرده اند و کسانی که کار را انجام می دهند. ۲) ویژگی های خود محصول و ۳) محیط پروژه که تیم نرم افزاری در آن کار می کند هنگامی که یک مدل فرآیند انتخاب گردید، تیم یک طرح پروژه مقدماتی براساس مجموعه فعالیت های چارچوبی فرآیند مشخص می کند هنگامی که طرح مقدماتی را ارائه شد، تجزیه فرآیند آغاز می شود یعنی یک طرح کامل، که وظایف کاری لازم برای پر کردن فعالیت های چارچوبی را منعکس می کند، باید ایجاد گردد. در بخش هایی که به دنبال خواهد آمد، این فعالیت ها را به اختصار مورد کاوش قرار می دهیم.

۳-٤-۱ تلفیق کردن فرآیند و محصول

طرح ریزی پروژه با تلفیق کردن محصول و فرآیند آغاز می شود هر عملی که قرار است توسط تیم نرم افزاری مهندسی شود، باید از میان یک سری فعالیت های چارچوبی بگذرد که برای یک سازمان نرم افزاری تعیین شده اند. فرض کنید که سازمان مجموعه فعالیت های چارچوبی زیر را برگزیده باشد.

- **ارتباط با مشتری**. وظایف لازم برای روشن شدن موثر نیازمندی ها بین سازنده و مشتری
- **طرح ریزی**. وظایف لازم برای تعیین منابع، خطوط زمانی و اطلاعات دیگر مربوط به پروژه
- **تحلیل ریسک**. وظایف لازم برای ارزیابی ریسک های مدیریتی و فنی
- **مهندسی** وظایف لازم برای ساخت یک یا چند شکل ارائه کاربرد
- **ساخت و ارائه**. وظایف لازم برای ساختن آزمودن، نصب کردن و ارائه خدمات پشتیبانی به مشتری (مثل مستندسازی و آموزش).
- **ارزیابی مشتری**. وظایف لازم برای به دست آوردن بازخورد مشتری براساس ارزیابی آشکال ارائه شده نرم افزار که طی مرحله مهندسی ایجاد و طی مرحله نصب، پیاده سازی شده اند.

اعضای تیم که روی هر یک از اعمال کار می کنند، همه فعالیت های چارچوبی را روی آن اعمال می کنند در اصل ماتریسی مشابه شکل ۲-۳ ایجاد می شود هر یک از اعمال اصلی محصول در ستون سمت چپ، لیست شده است فعالیت های چارچوبی در ردیف بعدی وارد می شوند کار مدیر پروژه (و دیگر اعضای تیم) عبارت است از برآورد نیازمندی های منابع برای هر خانه ماتریس، تاریخ های آغاز و پایان برای وظایف مرتبط با هر خانه و محصولات کاری که باید به عنوان نتیجه ای از هر خانه تولید شود.

۳-٤-۲ تجزیه فرآیند

یک تیم نرم افزار باید الگوی مهندسی نرم افزاری را انتخاب کند که برای وظایف مهندسی نرم افزار و پروژه، بهتر از همه باشد و به محض انتخاب شدن، مدل فرآیند را پر کند، دارای انعطاف پذیری زیاد باشد، یک پروژه نسبتا کوچک که مشابه کوشش های قبلی است به بهترین وجه با استفاده از روش ترتیبی خطی قابل انجام است اگر محدودیت های زمانی شدید تحمیل شده باشد و مسئله به طور کامل قابل تقسیم باشد، مدل RAD احتمالا گزینه مناسبی خواهد بود. اگر مهلت چنان کم باشد که عملکردهای کامل را نتوان به طور منطقی ارائه داد، یک راهبرد گام به گام احتمالا بهتر از همه است به طور مشابه پروژه هایی با ویژگی های دیگر (مانند خواسته های غیر قطعی، کشفیات جدید در فناوری مشتریان دشوار، پتانسیل استفاده مجدد) منجر به انتخاب مدل های فرآیند دیگری شود.

FIGURE 3.2
Melding the Problem and the Process

Common process framework activities	Customer communication	Planning	Risk analysis	Engineering	
Software engineering tasks					
Product functions					
Text input					
Editing and formatting					
Automatic copy edit					
Page layout capability					
Automatic indexing and TOC					
File management					
Document production					

شکل ۲-۳ تلفیق کردن مسئله و پروژه

هنگامی که مدل فرآیند انتخاب شد، چارچوب فرآیند مشترک، بر آن مطابقت داده می شود در هر مورد CPF بحث شده در این فصل ارتباط با مشتری، طرح ریزی، تحلیل ریسک، مهندسی، ساخت و ارائه و ارزیابی مشتری را می توان با الگو انطباق داد این چارچوب برای مدل های خطی، مدل های تکراری و گام به گام، برای مدل های ارزیابی و حتی برای مدل های مونتاژ مولفه ها یا همزمانی پاسخ می دهد CPF طبیعتی ثابت دارد و به عنوان مبنایی برای همه کارهای نرم افزاری انجام شده توسط یک سازمان نرم افزاری عمل می کند.

ولی وظایف کاری واقعی تغییر پذیرند، تجزیه فرآیند هنگامی آغاز می شود که مدیران پروژه بپرسند این فعالیت CPF را چگونه آغاز می کنیم برای مثال یک پروژه کوچک و نسبتا ساده ممکن است برای فعالیت برقراری ارتباط با مشتری نیاز به وظایف کاری زیر داشته باشد:

۱. تهیه لیستی از مسائل مربوط به روشن کردن امور
۲. ملاقات با مشتری برای پرداختن به مسائل مربوط به روشن کردن امور
۳. تهیه بیان حوزه
٤. مرور بیان حوزه با همه دست اندرکاران
۵. اصلاح بیان حوزه در صورت نیاز.

این رویدادها ممکن است طی یک دوره زمانی کمتر از ٤٨ ساعت به وقوع بپیوندد. این ها نشان دهنده تجزیه فرآیندی هستند که برای پروژه کوچک و نسبتا ساده مناسبند.

اکنون پروژه پیچیده تری را در نظر می گیریم که حوزه ای گسترده تر داشته تاثیر آن بر بازار کار چشمگیرتر است. چنین پروژه ای ممکن است برای فعالیت برقراری ارتباط با مشتری نیاز به وظایف کاری زیر داشته باشد:

۱. بازبینی درخواست مشتری
۲. طرح ریزی و زمان بندی یک ملاقات رسمی با مشتری

۳. پژوهش برای تعیین راهکار پیشنهادی و روش های موجود
٤. آماده سازی یک سندکاری و برنامه برای ملاقات رسمی
۵. اجرای ملاقات
٦. تهیه مشخصات جزئی که ویژگی های رفتاری، عملیاتی، و داده ای نرم افزار را منعکس کند.
۷. بازبینی هر مشخصات جزئی برای تصحیح، سازگاری و عدم ابهام.
۸. مونتاژ مشخصات جزئی در مستندات تعیین حوزه
۹. بازبینی مستندات تعیین حوزه
۱۰. اصلاح مستندات تعیین حوزه بنا به نیاز

هر دو پروژه یک فعالیت چارچوبی انجام می دهند که آن را برقراری ارتباط با مشتری می نامیم، ولی تیم پروژه اولی نیمی از وظایف کاری مهندسی نرم افزار پروژه دوم را انجام می دهند.

۳-۵ پروژه

برای مدیریت یک پروژه نرم افزاری موفق، باید بدانیم چه چیزهایی ممکن است اشتباه شود (به طوری که بتوان از مشکلات بر حذر بود) و چگونه می توان آن را به راحتی انجام داد. جان ریل در یک مقاله عالی درباره پروژه هایی نرم افزاری ده علامت را مشخص می کند که نشان می دهد پروژه سیستم های اطلاعاتی در معرض خطر قرار دارد.

۱. افراد نرم افزاری نیازهای مشتری را درک نمی کنند.
۲. حوزه محصول، خوب تعیین نشده است.
۳. تغییرات خوب مدیریت نمی شوند
٤. فناوری انتخاب شده تغییر می کند
۵. نیازهای تجاری تغییر می کند (یا خوب تعریف نمی شوند).
٦. مهلت ها واقع بینانه نیست.
۷. کاربران مقاومت می کنند.
۸. حمایت مالی از بین می رود یا هرگز به طور مناسب به دست نیامده است.
۹. تیم پروژه فاقد افرادی با مهارت های لازم است.
۱۰. مدیران و سازندگان از بهترین کوشش ها و دروس فراگرفته شده پرهیز می کنند.

حرف های مجربین صنعت، غالبا هنگام بحث درباره پروژه های نرم افزاری بسیار دشوار به قاعده ۹۰-۹۰ رجوع می کنند: ۹۰٪ اول یک سیستم ۹۰٪ زمان و کار را به خود اختصاص می دهد مقدمات قاعده ۹۰-۹۰ را می توان در علائم ذکر شده در لیست بالا یافت.

ولی منفی بافی کافی است یک مدیر برای پرهیز از مشکلات فوق چگونه عمل می کند؟ ریل یک روش پنج مبتنی بر عقل سلیم را برای پروژه های نرم افزاری پیشنهاد می کند.

در آغاز کار درست گام بردارید. برای این منظور باید سخت کار کرد تا مسئله مورد نظر به درستی درک شود و سپس برای همه کسانی که در پروژه دخالت دارند، انتظارات و اهداف واقع بینانه بنا کرد، این کار با یک ساختار تیمی مناسب و دادن خود مختاری مسئولیت و فناوری مورد نیاز به تیم تقویت می شود.

تکانه را حفظ کنید. بسیاری از پروژه ها شروع بسیار خوبی دارند ولی بعد به آهستگی زوال می یابند برای حفظ تکانه مدیر پروژه باید انگیزه هایی جهت حفظ توان عملیاتی افراد در یک مقدار حداقل مطلق فراهم آورد تیم باید در هر وظیفه ای که انجام می دهد بر کیفیت تاکید ورزد و مدیر ارشد باید هر چه می تواند انجام دهد تا سرراه تیم مانع ایجاد نکند.

پیشرفت کار را دنبال کنید. برای یک پروژه نرم افزاری پیشرفت به موازات تولید محصولات کاری (مثلا مشخصات، کدمنبع، مجموعه موارد آزمایشی) دنبال می شود و به عنوان بخشی از فعالیت تضمین کیفیت (با استفاده از بازبینی های فنی رسمی) مورد تایید قرار می گیرد به علاوه فرآیند نرم افزار و سنجش های پروژه را می توان جمع آوری کرد و برای ارزیابی پیشرفت در برابر میانگین های تهیه شده برای سازماندهی توسعه نرم افزار به کار گرفت.

هوشمندانه تصمیم گیری کنید. در اصل تصمیم گیری های مدیریت پروژه و تیم نرم افزار باید در جهت ساده کردن امور باشند هر گاه ممکن بود، تصمیم بگیرید از مولفه های نرم افزاری موجود یا آماده استفاده نمایید در صورت در دسترس بودن روش های استاندارد تصمیم بگیرید که از واسط های سفارشی پرهیز کنید، تصمیم بگیرید به وظایف پیچیده یا خطرناک بیش از آن چیزی که فکر می کنید، زمان اختصاص دهید.

تحلیلی پسانوگرایانه ارائه دهید، راهکاری ارائه دهید که از تجربیاتی که از پروژه کسب کردید، استفاده نمایند، زمان بندی طرح ریزی شده واقعی را ارزیابی کنید، معیارهای پروژه نرم افزاری را جمع آوری و تحلیل کنید، بازخورد اعضای تیم و مشتریان را کسب کنید و این یافته ها را به صورت مکتوب ثبت نمایید.

۶-۳ اصل W^5HH

باری بوهم در یک مقاله عالی درباره پروژه ها و فرآیند نرم افزار می گوید:" به یک اصل سازماندهی نیاز داید تا در مقیاس کوچک، طرح های ساده ای برای پروژه های ساده مشخص کند"، بوهم روشی پیشنهاد می کند که اهداف پروژه، نقاط عطف و زمانبندی ها، مسئولیت ها روش فنی و مدیریتی، و منابع مورد نیاز را دربر می گیرد. وی این روش را اصل WWWWWHH می نامد پس از یک سری پرسش ها، منجر به تعریف ویژگی های کلیدی پروژه و طرح پروژه می شود.

این سیستم چرا توسعه می یابد؟ پاسخ این پرسش همه افراد را قادر به ارزیابی اعتبار دلایل تجارتی کار نرم افزار می کند به بیان دیگر، آیا هدف تجاری صرف کردن زمان و هزینه را توجیه می کند؟

چه چیزی و در چه زمانی انجام خواهد شد؟ برای پاسخ دادن به این پرسش ها به تیم کمک کنید تا با شناسایی وظایف کلیدی پروژه و نقاط عطف که مورد نیاز مشتری هستند یک برنامه زمانبندی برای پروژه وضع کند.

چه کسی مسئول یک عمل است؟ پیش از این در همین فصل متذکر شدیم که که نقش و مسئولیت هر عضو تیم نرم افزاری باید مشخص شود پاسخ این پرسش، منظور فوق را برآورده می سازد.

از لحاظ سازمانی چه جایگاهی دارند؟ همه وظایف و مسئولیت ها در خود تیم نرم افزار باقی نمی ماند، مشتری، کاربران و واگذارندگان دیگر نیز مسئولیت های دارند.

کار از نظر مدیریتی و فنی چگونه انجام خواهد شد؟ هنگامی که دامنه کاربرد محصول معین شد، باید یک راهبر مدیریتی و فنی برای پروژه تعریف شود.

از هر منبع چه میزان لازم است؟ پاسخ این پرسش از برآوردهایی مبتنی بر پاسخ پرسش های قبلی به دست می آید.

فصل ٤ : معیارهای پروژه و فرآیند نرم افزار

اندازه گیری چیست؟ معیارهای محصول و فرآیند نرم افزاری مقادیری کمی هستند که درک بازدهی فرآیند نرم افزار و پروژه هایی را امکان پذیر می سازند که از آن فرآیند به عنوان چارچوب کار استفاده می کنند سپس این داده ها تحلیل می شوند، با میانگین های گذشته مقایسه می شوند و مورد ارزیابی قرار می گیرند تا معین شود که آیا بهبود بهره وری و کیفیت رخ داده است یا خیر. معیارها برای مسائل و نکات ظریف نیز به کار می روند به طوری که بتوان درمانی برای آنها یافت و فرآیند نرم افزار را بهبود بخشید.

چه می کند؟ معیارهای نرم افزاری توسط مدیران نرم افزار تحلیل و ارزیابی می شوند میزان ها غالبا توسط مهندسان نرم افزار جمع آوری می شوند.

چرا اهمیت دارد؟ اگر اندازه گیری نکنید، داوری شما فقط مبتنی بر ارزیابی موضوعی خواهد بود با اندازه گیری روند امور (چه خوب چه بد) را می توان مشخص کرد برآوردها را بهتر می توان انجام داد، و با گذشت زمان می توان به بهبود واقعی دست یافت.

مراحل کار کدام است؟ با تعریف یک مجموعه محدود از میزان های فرآیند، پروژه و محصول شروع کنید که جمع آوری آنها آسان است این میزان ها غالبا با استفاده از معیارهای اندازه گرا یا عمل گرا به هنجار می شوند نتیجه تحلیل می شود و با میانگین های گذشته برای پروژه های مشابه انجام شده در همان سازمان، مقایسه می شوند، روندها ارزیابی می شود و نتایج استنباط می شود.

محصول کار چیست؟ مجموعه ای از معیارهای نرم افزاری که منجر به درک فرآیند و پروژه می شود

چگونه مطمئن شوم که درست از عهده امور برآمده ام؟ با اعمال یک طرح اندازه گیری سازگار و در عین حال ساده که هرگز برای ارزیابی، تشویق یا تنبیه کارآیی فردی به کار نمی رود.

در قاموس مدیریت پروژه های نرم افزاری، در وهله نخست، یا معیارهای بهره وری و کیفیت سر و کار خواهیم داشت موازین توسعه نرم افزاری به صورت تابعی از انرژی و زمان صرف شده و موازین آمادگی استفاده برای محصولات کاری تولید شده بیان می شود.

پارک، گوترت و فلوراک دلایل اندازه گیری را در کتاب خود در باب اندازه گیری نرم افزار مورد بحث قرار داده اند:
چهار دلیل برای اندازه گیری فرآیندهای نرم افزاری، محصولات و منابع وجود دارد:

- مشخص کردن ویژگی ها
- ارزیابی کردن
- پیش بینی کردن
- بهبود بخشیدن

ویژگی های فرآیندها، محصولات، منابع و محیط ها را مشخص می کنیم تا آنها را درک کنیم و مبناهایی برای مقایسه ارزیابی های آینده بنا کنیم.

ارزیابی می کنیم تا وضعیت طرح ها را معین کنیم. موازین، حسگرهایی هستند که به ما اطلاع می دهند فرآیندها و پروژه های ما چه زمانی از مسیر اصلی خود منحرف می شوند و می توانیم آنها را دوباره تحت کنترل در آوریم. همچنین عمل ارزیابی را انجام می دهیم تا بتوانیم میزان دستیابی به اهداف کیفیتی و تاثیرات بهبود فناوری و فرآیند بر محصول را بسنجیم.

پیش بینی را بدین منظور انجام می دهیم تا بتوانیم طرح ریزی کنیم، اندازه گیری برای پیش بینی، شامل درک روابط میان فرآیندها و محصولات و ساخت مدل هایی از این روابط می شود، به طوری که مقادیر مشاهده شده برای برخی صفات را بتوان برای پیش بینی صفات دیگر به کار برد این کار را از آنرو انجام می دهیم که می خواهیم اهداف قابل دستیابی برای هزینه، زمانبندی و کیفیت بنا کنیم به طوری که منابع مناسبی را بتوان به کار برد اندازه گیری برای پیش بینی اساس برون یابی روند امور را نیز تشکیل می دهد از این رو، برآورد هزینه زمان و کیفیت را می توان براساس شواهد جاری به هنگام سازی کرد برآوردها و تخصیص های مبتنی بر داده های قبلی نیز به تحلیل خطرات و برقراری مطالحه میان طراحی و هزینه کمک می کند.

اندازه گیری می کنیم تا زمان جمع آوری اطلاعات کمی را بهبود بخشیم و به این ترتیب به شناسایی موانع، علل ریشه ای و عدم بازدهی کمک می شود.

٤-١ موازین، معیارها و شاخص ها

گرچه واژه های میزان، اندازه گیری و معیار غالبا مترادف بوده و به جای هم به کار می روند، توجه به تفاوت های ظریف میان آنها اهمیت دارد در حیطه مهندسی نرم افزار، میزان عبارت از کمیتی است که نشانگر حد، مقدار، ابعاد، ظرفیت یا اندازه صفتی از محصول یا فرآیند است اندازه گیری عمل تعیین میزان است در فرهنگ واژه های مهندسی نرم افزار IEEE معیار چنین تعریف می شود «میزان کمی از حدی که یک سیستم، مولفه یا فرآیند می تواند دارای یک صفت مفروض باشد».

هنگام که یک نقطه داده ای منفرد جمع آوری شده باشد (مثل تعداد خطاهای کشف شده در بازبینی یک پیمانه منفرد) یک میزان بنا شده است، اندازه گیری در نتیجه جمع آوری میزان هایی از تعداد خطاها برای هر مورد رخ می دهد معیار نرم افزاری به نحوی با تک تک میزان ها در ارتباط است (مثل تعداد میانگین خطاهای یافت شده به ازای هر بازبینی یا تعداد میانگین خطاهای یافت شده به ازای هر نفر ساعت صرف شده روی بازبینی ها).

مهندس نرم افزار معیارهای را جمع آوری و ایجاد می کند، به طوری که شاخص هایی از آنها به دست می آید، شاخص، معیار یا ترکیبی از معیارهاست که درکی از فرآیند نرم افزار، پروژه نرم افزاری یا خود محصول به دست می دهد شاخص، دیدی فراهم می آورد که مدیر پروژه یا مهندسان نرم افزار را قادر به تنظیم فرآیند می سازد تا شرایط پروژه یا فرآیند بهتر شود.

برای مثال، چهار تیم نرم افزاری روی یک پروژه نرم افزاری بزرگ مشغول کارند هر یک از تیم ها باید طراحی را مورد بازبینی قرار دهد، ولی مجاز به گزینش نوع بازبینی مورد استفاده است با بررسی معیار (خطاهای یافت شده به ازای هر نفر –ساعت) مدیر پروژه متوجه می شود که هر دو تیمی که از روش های بازبینی رسمی تر استفاده می کند، معیار مذکور ٤٠٪ بیشتر از تیم های دیگر است با فرض آنکه تمامی پارامترهای دیگر مساوی باشند، مدیر پروژه شاخصی به دست می آورد مبنی بر این که روش های بازبینی رسمی ممکن است نسبت به روش های غیر رسمی به زمان بیشتری نیاز داشته باشند، مدیر ممکن است پیشنهاد کند که تیم ها از روش رسمی تر استفاده کنند. این معیار به مدیر چنین درکی داده است و درک، منجر به تصمیم گیری آگاهانه می شود.

٤-٢ معیارها در دامنه فرآیند و پروژه

اندازه گیری در جهان مهندسی بسیار متداول است، مصرف برق، وزن، ابعاد فیزیکی، دما، ولتاژ، نسبت به سیگنال به نویز و.... همگی از مواردی هستند که اندازه می گیریم و البته این لیست همچنان می تواند ادامه یابد

معیارها را باید طوری جمع آوری کرد که شاخص های فرآیند و محصول را بتوان کشف کرد شاخص های فرآیند به سازمان مهندسی نرم افزار این امکان را می دهند که از بازدهی فرآیند موجود (یعنی الگو، وظایف مهندسی نرم افزار، محصولات کاری و نقاط عطف) دیدی درست به دست آورد. مدیران و سازندگان را قادر می سازند تا بتوانند ارزیابی کنند چه چیزهایی عملی هستند و چه چیزهایی نیستند معیارهای فرآیند، از همه پروژه ها و در مدت زمانی طولانی جمع آوری می شوند هدف از استفاده آنها فراهم آوردن شاخص هایی است که منجر به بهبود فرآیند نرم افزار در بلند مدت می شود.

شاخص های پروژه مدیر پروژه را قادر می سازند تا ۱) به وضعیت پروژه در حال پیشرفت دست پیدا کند، ۲) خطرات بالقوه را دنبال کند. ۳) نواحی مشکل آفرین را پیش از بحرانی شدن آنها کشف کند. ٤) جریان کار یا وظایف را تنظیم کند و ۵) توانایی تیم پروژه در کنترل کیفیت محصولات کاری نرم افزار را ارزیابی کند.

در برخی موارد، برخی معیارهای نرم افزار را می توان برای تعیین شاخص های پروژه و سپس فرآیند به کار برد. در واقع معیارهایی که توسط یک تیم پروژه جمع آوری و به معیارهایی برای استفاده در طی پروژه تبدیل می شوند، قابل انتقال به کسانی هستند که مسئول بهبود بخشیدن به فرآیند نرم افزار هستند به این دلیل، بسیاری از معیارهای یکسان در هر دو دامنه فرآیند و پروژه استفاده می شوند.

١-٢-٤ معیارهای فرآیند و بهبود فرآیند نرم افزار

تنها راه موجه برای بهبود بخشیدن به هر فرآیندی، اندازه گیری صفات مشخصی از آن فرآیند توسط یک مجموعه معیارهای با معنی براساس این صفات و سپس استفاده از این معیارها برای فراهم آوردن شاخص هایی است که منجر به راهبردی برای بهبود کار می شود ولی پیش از بحث درباره معیارهای نرم افزاری و تاثیر آنها بر بهبود فرآیند نرم افزار، توجه به این نکته حائز اهمیت است که فرآیند، فقط یکی از چند عامل قابل کنترل در بهبود بخشیدن کیفیت نرم افزار و کارآیی سازمانی است.

شکل ١-٤ نشان می دهد که فرآیند، نقطه مرکزی مثلثی است که سه عامل مهم تاثیر گذار بر کیفیت نرم افزار و کارآیی سازمانی را به هم متصل می کند. ثابت شده است که انگیزش و مهارت افراد، موثرترین عامل در کیفیت و کارآیی است پیچیدگی محصول می تواند اثری چشمگیر بر کیفیت و کارآیی تیم بگذارد. فناوری (یعنی روش های مهندسی نرم افزار) که فرآیند را تشکیل می دهد نیز موثر است به علاوه مثلث فرآیند در داخل دایره ای از شرایط محیطی قرار گرفته است که شامل محیط توسعه (یعنی ابزار کیس) شرایط بازار کار (مثل مهلت ها و قواعد تجاری) و خصوصیات مشتری (مثل سهولت برقراری ارتباط) می شود.

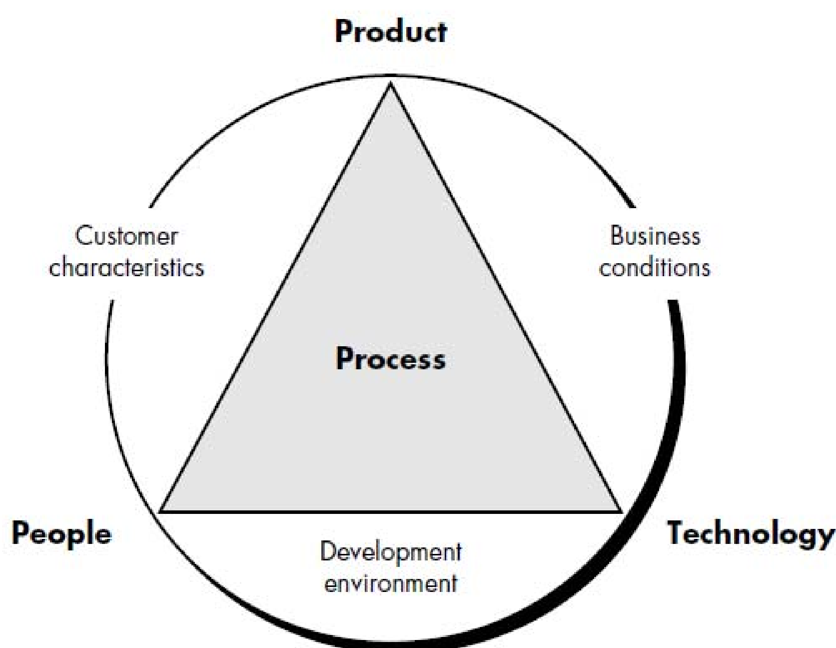
بازدهی یک فرآیند نرم افزاری را به طور غیر مستقیم اندازه گیری می کنیم یعنی یک مجموعه معیار براساس نتایجی به دست می آوریم که از فرآیند قابل حصول هستند نتایج شامل موازین خطاهای کشف شده، پیش از ارائه نرم افزار، نقایص گزارش شده توسط کاربران نهایی، محصولات کاری تحویل شده (بهره وری)، انرژی انسانی صرف شده، زمان صرف شده، مطابقت با برنامه زمانبندی و موازین دیگر می شود همچنین معیارهای فرآیند توسط اندازه گیری ویژگی های وظایف خاص مهندسی نرم افزار به دست می آوریم، برای مثال ممکن است انرژی زمان صرف شده در اجرای فعالیت های پوششی و فعالیت های کلی مهندسی نرم افزار را اندازه گیری می کنیم.

گریدی استدلال می کند که برای انواع متفاوت داده های فرآیند، کاربردهای خصوصی و عمومی وجود دارد. از آنجا که طبیعتا مهندسان نرم افزار ممکن است به استفاده از معیارهای جمع آوری شده بر مبنایی فردی حساس باشند این

فلسفه داده های فرآیند خصوصی به خوبی با روش فرآیند نرم افزار شخصی پیشنهاد شده توسط همفری مطابقت دارد همفری این روش را به این طریق زیر شرح می دهد:

فرآیند نرم افزار شخصی (PSP) مجموعه ای ساخت یافته از توصیفات فرآیند اندازه گیری ها و روشهایی است که می تواند به مهندسان کمک کند تا کارآیی شخصی خویش را بهبود بخشند، این فرآیند فرم ها و نوشتارها و استاندارهایی را فراهم می آورد که در برآورد و طرح ریزی کارشان کمک می کند به آنها نشان می دهد چگونه فرآیندها را تعریف کنند و کیفیت و بهره وری فرآیندها را چگونه تعیین کنند یک اصل بنیادین PSP آن است که افراد با هم متفاوتند و روشی که برای یک مهندس موثر است ممکن است برای دیگری مناسب نباشد از اینرو، PSP به مهندسان در اندازه گیری و دنبال کردن کارشان کمک می کند، به طوری که بتواند روش هایی بیابند که برای آنها بهترین باشد.

همفری می داند که بهبود فرآیند نرم افزار می تواند و باید در سطحی فردی آغاز گردد داده های فرآیند خصوصی می توانند به عنوان محرکی مهم در راستای بهبود کار مهندس نرم افزاری که تنها کار می کند عمل کنند.



شکل ۱-٤ مواردی که کیفیت نرم افزار و کارآیی سازمانی را تعیین می کنند.

برخی معیارهای فرآیند، برای تیم پروژه نرم افزار، خصوصی به شمار می روند ولی در قبال همه اعضای تیم، عمومی هستند مثال ها شامل نقایص گزارش شده برای اعمال اصلی نرم افزار (که توسط چند تن از سازندگان، توسعه یافته اند) خطاهای یافته شده در اثنای بازبینی های فنی رسمی، تعدا خطوط کد یا نقاط تابع به ازای هر پیمانه یا هر عمل هستند این داده ها توسط تیم مورد بازبینی قرار می گیرند تا شاخصهایی برای بهبود بخشیدن به کارآیی تیم به دست آید.

معیارهای عمومی عموما اطلاعاتی را جذب می کنند که در آغاز برای افراد و تیم ها جنبه خصوصی داشته اند تعداد نقایص در سطح پروژه (که مطلقا به یک فرد نسبت داده نمی شود) انرژی، زمان و داده های مربوط، جمع آوری و ارزیابی می شوند تا شاخص هایی که می توانند کارآیی فرآیند سازمانی را بهبود بخشند کشف شوند.

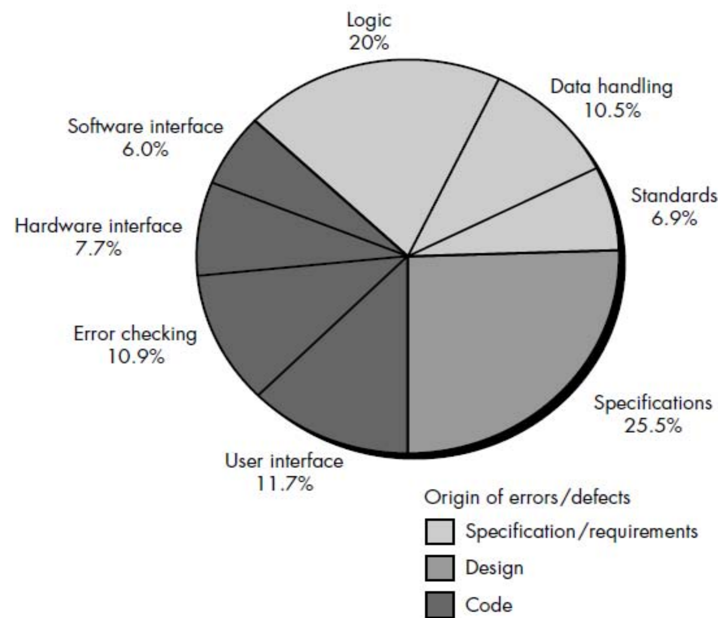
معیارهای فرآیند نرم افزار می توانند به موازاتی که یک سازمان در جهت بهبود بخشیدن به سطح کلی بلوغ فرآیند عمل می کند، مزایای چشمگیری به همراه داشته باشند، ولی همانند همه معیارها، امکان استفاده نادرست از آنها نیز وجود دارد و ممکن است بیش از آنکه مسئله را حل کنند، بر مشکلات بیفزایند. گریدی یک اتیکت معیار نرم افزار را پیشنهاد می کند که مدیران برای ایجاد برنامه معیارهای فرآیند از آن استفاده می کنند.

- استفاده از عقل سلیم و حساسیت سازمانی به هنگام تفسیر داده های معیاری.
- تهیه بازخورد منظمی از افراد و تیم هایی که برای جمع آوری موازین و معیارها کار کرده اند.
- عدم استفاده از معیارها برای ارزیابی افراد.
- کار با سازندگان و تیم ها برای تعیین اهداف واضح و معیارهایی که برای رسیدن به آنها مورد استفاده قرار خواهند گرفت.
- عدم استفاده از معیارها برای تهدید افراد یا تیم ها.
- داده های معیاری که نشانگر یک زمینه مشکل دار هستند نباید منفی انگاشته شوند این داده ها فقط شاخصی برای بهبود فرآیند هستند.
- تنها به یک معیار تکیه نکنید تا معیارهای مهم دیگر از قلم نیفتند.

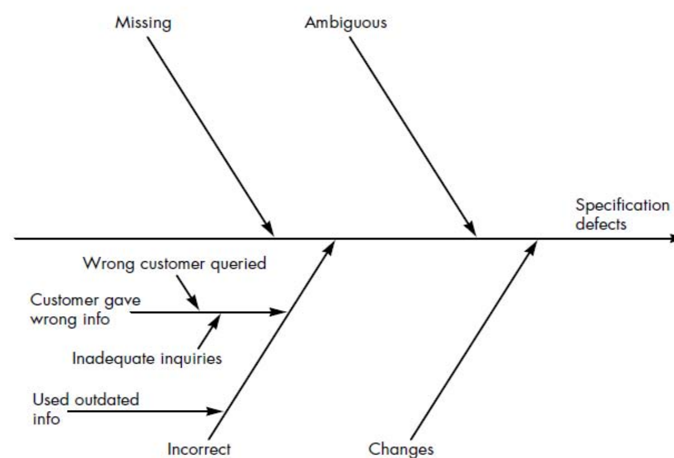
هر چه سازمانی در جمع آوری و استفاده از معیارهای فرآیند، احساس راحتی بیشتری داشته باشد، بدست آوردن شاخص های ساده، روش دقیق تر مرسوم به بهبود آماری فرآیند نرم افزاری را نتیجه می دهد در اصل SSPI از تحلیل شکست نرم افزار برای جمع آوری داده های مربوط به همه خطاها و نقایص مشاهده شده در اثنای توسعه و بکارگیری سیستم یا محصول استفاده می کند تحلیل شکست به شیوه زیر صورت می پذیرد.

۱. همه خطاها و نقایص برحسب منشا دسته بندی می شوند (مثل نقص در مشخصات، نقص در منطق، عدم تطابق با استانداردها).
۲. هزینه تصحیح هر خطا و نقص ثبت می شود.
۳. تعداد خطاها و نقایص در هر دسته شمارش و به ترتیب نزولی مرتب می شوند.
٤. هزینه کلی خطا و نقایص در هر گروه محاسبه می شود.
٥. داده های حاصل مورد تحلیل قرار می گیرند تا دسته هایی که منجر به بالاترین هزینه ها در سازمان می شوند کشف شوند.
٦. تدابیری برای اصلاح فرآیند با هدف حذف (یا کاهش فراوانی رخ دادن) دسته ای از خطاها و نقایص که بیشترین هزینه را دارند اندیشیده می شود.

با دنبال کردن مراحل ۱و ۲ در بالا توزیع ساده ای از نقایص به دست می آید (شکل۲-٤). برای نمودار دایره ای نشان داده شده در شکل، هشت نقص و منشا آنها داده شده است گریدی توسعه یک نمودار استخوان ماهی را برای کمک به بررسی داده های ارائه شده در نمودار فراوانی پیشنهاد می کند در شکل ۳-٤، ستون فقرات نمودار (خط مرکزی) نشانگر عامل کیفیت مورد نظر است، (در این مورد نقایص مشخصات که ۲۵٪ از کل نقایص را تشکیل می دهد) هر یک از دنده ها (خطوط مورب) که به ستون فقرات متصل است نشانگر علل بالقوه برای مشکل کیفیتی (از قبیل کمبود خواسته ها، مشخصه مبهم، خواسته های نادرست، تغییر خواسته ها) است. سپس ستون فقرات و دنده ها به هر یک از



شکل ۲-٤ علل نقایص و منشاءِ آن برای چهار پروژه نرم افزاری



شکل ۳-٤ نمودار استخوان ماهی (برگرفته از [GRA92]).

جمع آوری معیارهای فرآیند، محرکی برای خلق نمودار استخوان ماهی است با تحلیل یک نمودار استخوان ماهی کامل می توان شاخص هایی به دست آورد که سازمان نرم افزاری را قادر به اصلاح فرآیند در جهت کاهش دادن فرکانس خطاها و نقایص می سازد.

۴-۲-۲ معیارهای پروژه

معیارهای فرآیند نرم افزاری برای اهداف راهبردی بکار می روند، معیارهای پروژه نرم افزاری، تاکتیکی هستند مدیر پروژه و تیم نرم افزاری برای تطبیق جریان کار پروژه و فعالیت های فنی از معیارهای پروژه و شاخص های دست آمده از آنها استفاده می کند.

نخستین کاربرد معیارهای پروژه در اکثر پروژه های نرم افزاری، طی برآورد رخ می دهد معیارهای جمع آوری شده از پروژه های گذشته، به عنوان مبنایی برای برآورد زمان و انرژی صرف شده جهت کار نرم افزاری کنونی استفاده می شود به موازات پیشرفت پروژه، موازین انرژی و زمان صرف شده با برآوردهای اولیه (و زمانبندی پروژه) مقایسه می شوند مدیر پروژه از این داده ها برای نظارت و کنترل پیشرفت کار استفاده می کند.

با شروع کارهای فنی، معیارهای دیگر پروژه رفته رفته اهمیت پیدا می کنند آهنگ تولید برحسب تعداد صفحات مستندات، ساعت های بازبینی نقاط تابع، و تعداد خطوط منبع تحویل شده اندازه گیری می شود به علاوه خطاهای کشف شده در اثنای انجام هر یک از وظایف مهندسی نرم افزار پیگیری می شوند به موازات تکامل نرم افزار از مشخصات به طراحی، معیارهای فنی جمع آوری شده، در ارزیابی کیفیت طراحی و فراهم آوردن شاخص هایی که در روش اتخاذ شده برای تولید و آزمایش کد تاثیر دارند، مورد استفاده قرار می گیرند.

معیارهای پروژه هدفی دو گانه دارند، نخست، این معیارها برای به حداقل رساندن زمانبندی توسعه (با انجام تنظیمات لازم برای پرهیز از تاخیرها و کاهش دادن مشکلات و خطرات بالقوه) به کار می روند، دوم معیارهای پروژه برای ارزیابی کیفیت محصول بر پایه پیشرفت کار و در صورت نیاز، برای اصلاح روش فنی جهت بهبود بخشیدن به کیفیت به کار می روند.

به موازاتی که کیفیتی بهبود می یابد، نقایص به حداقل می رسند و با کم شدن شمار نقایص، میزان دوباره کاری در اثنای پروژه نیز کاهش می یابد، این کار منجر به کاهش هزینه کل پروژه می شود.

در یک مدل دیگر برای معیارهای پروژه نرم افزاری، پیشنهاد می شود که هر پروژه باید موارد زیر را اندازه گیری کند

- **ورودی ها.** میزان هایی از منابع (مثل افراد، محیط) که برای انجام کار لازم است.
- **خروجی ها**، میزان هایی از قطعات قابل تحویل یا محصولات کاری ایجاد شده طی فرآیند مهندسی نرم افزار.
- **نتایج**. میزان هایی که نشان دهنده میزان کارآمدی قطعات تحویل شده اند

درواقع، این مدل را هم در مورد پروژه و هم فرآیند می توان به کاربرد، در قلمرو پروژه، این مدل را می توان به طور بازگشتی به موازات هر یک از فعالیت ها چارچوبی به کاربرد، بنابراین، خروجی های حاصل از یک فعالیت به ورودی های فعالیت بعدی تبدیل می شوند معیارهای نایج را می توان برای فراهم آوردن شاخصی از مفید بودن محصولات کاری، به موازات حرکت از یک فعالیت چارچوبی به فعالیت دیگر، به کاربرد.

۴-۳ اندازه گیری نرم افزار.

اندازه گیری های جهان فیزیکی را می توان به دو شیوه گروه بندی کرد موازین مستقیم (مانند طول یک پیچ) و موازین غیر مستقیم (مانند کیفیت پیچ های تولید شده که با سفارش تعداد نمونه های رد شده سنجیده می شود) معیارهای نرم افزاری را می توان به طور مشابه گروه بندی کرد.

موازین مستقیم فرآیند مهندسی نرم افزار، شامل هزینه و انرژی به کار رفته است موازین مستقیم محصول، شامل تعداد خطوط کد، (LOC) تولید شده، سرعت اجرا، اندازه حافظه و نقایص گزارش شده در یک مدت زمان تعیین شده است موازین غیر مستقیم محصول عبارتند از : عملکرد، کیفیت، پیچیدگی، بازدهی، و عملکرد نرم افزار یا بسیاری از قابلیت های دیگر.

مادامی که قراردادهای مشخصی از قبل وضع شده باشد، هزینه و انرژی لازم برای ساخت نرم فزار، تعداد خطوط کد تولید شده و موازین مستقیم را به راحتی می توان به دست آورد. ولی ارزیابی کیفیت و عملکرد نرم افزار یا بازدهی یا قابلیت نگهداری دشوارتر است و فقط به طور غیر مستقیم قابل اندازه گیری است.

پیش از این، دامنه معیارهای نرم افزاری را به معیارهای فرآیند، پروژه و محصول تقسیم کردیم، همچنین متذکر شدیم، معیارهایی از محصول که برای افراد جنبه خصوصی دارند غالبا ترکیب می شود و معیارهای پروژه را توسعه می دهند که برای تیم نرم افزاری جنبه عمومی دارند، سپس معیارهای پروژه تقویت می شوند تا معیارهای فرآیند ایجاد شوند که برای کل سازمان نرم افزاری جنبه عمومی دارند، ولی یک سازمان چگونه معیارهایی را که از افراد و پروژه های متفاوت به دست می آید، با هم ترکیب می کند؟

برای نشان دادن این موضوع، مثال ساده ای را در نظر می گیریم، افرادی که روی دو پروژه متفاوت کار می کنند همه خطاهایی که طی فرآیند نرم افزار می یابند، ثبت و گروه بندی می کنند سپس موازین فردی با هم ترکیب شده موازین تیمی را تشکیل می دهند تیم A پیش از ارائه نرم افزار به بازار و در اثنای فرآیند نرم افزار، ۳۴۲ خطا یافت. تیم B تعداد ۱۸۴ خطا یافت اگر همه چیزهای دیگر برابر فرض شود، کدام تیم در کشف خطاها در سرتاسر فرآیند کارآمدتر بوده است؟ چون ما اندازه یا پیچیدگی پروژه ها را نمی دانیم، نمی توانیم این پرسش را پاسخ بدهیم ولی اگر، میزان ها نرمال سازی شده باشند، امکان ایجاد معیارهای نرم افزاری را فراهم می آورد که مقایسه با میانگین های سازمانی وسیع تر را ممکن می سازند.

۱-۳-٤ معیارهای اندازه گرا

معیارهای اندازه گرا از نرمال سازی موازین کیفیتی و یا بهره وری، و با در نظر گرفتن اندازه نرم افزار تولید شده به دست می آیند اگر یک سازمان نرم افزاری رکوردهای ساده ای نگهداری کند، جدولی از موازین اندازه گرا نظیر شکل ٤-٤ ایجاد می شود در این جدول همه پروژه های توسعه نرم افزاری که طی چند سال گذشته به انجام رسیده اند و موازین مرتبط با آنها آورده شده است با رجوع به ردیف مربوط به پروژه آنها (شکل ٤-٤) مشاهده می شود که ۱۲۰۰۰ خط کد (LOC) با صرف ۲٤ نفر ماه انرژی و صرف هزینه ۱۶۸۰۰۰ پوند توسعه یافت. لازم به ذکر است که انرژی و هزینه ثبت شده در جدول، همه فعالیت های مهندسی نرم افزار (تحلیل، طراحی، کد و آزمایش) را نشان می دهد، نه فقط کدنویسی را اطلاعات دیگر برای پروژه آلفا نشان می دهد که ۳۶۵ صفحه مستندات ایجاد شده است. ۱۳٤ خطا پیش از ارائه نرم افزار به مشتری و ۲۹ خطا پس از ارائه در نخستین سال عملکرد آن ثبت شده است برای توسعه نرم افزار پروژه آلفا سه نفر کار کرده اند.

Project	LOC	Effort	$(000)	Pp. doc.	Errors	Defects	People
alpha	12,100	24	168	365	134	29	3
beta	27,200	62	440	1224	321	86	5
gamma	20,200	43	314	1050	256	64	6
•	•	•	•	•	•		
•	•	•	•	•	•		
•	•	•	•	•	•		

FIGURE 4.4
Size-oriented metrics

شکل ٤-٤ معیارهای اندازه گرا

برای توسعه معیارهایی که با معیارهای مشابه حاصل از پروژه های دیگر قابل امتزاج باشند، تعداد خطوط کد را به عنوان مقدار نرمال سازی انتخاب می کنیم با استفاده از داده های اساسی جدول، برای هر پروژه می توان مجموعه ای از معیارهای اندازه گیری ساده را توسعه داد:

- تعداد خطا به ازای هر KLOC(هزار خط کد)
- نقایص به ازای هر KLOC
- هزینه مصرفی به ازای هر LOC
- تعداد صفحات مستندات به ازای هر KLOC

معیارهای جالب دیگری را نیز می توان توسعه داد:

- تعداد خطاها به ازای هر نفر ماه
- LOC به ازای هر نفر ماه
- هزینه مصرفی به ازای هر صفحه از مستندات

٤-٣-٢ معیارهای عملکردگرا

معیارهای عملکردگرا از میزان عملکرد ارائه شده توسط برنامه کاربردی، به عنوان مقداری برای نرمال سازی استفاده می کنند چون عملکرد را مستقیما نمی توان اندازه گیری کرد باید آنرا به طور غیر مستقیم با استفاده از موازین مستقیم دیگر به دست آورد. معیارهای عملکردگرا نخستین بار توسط آلبرشت پیشهاد شدند او میزانی موسوم به نقطه عملکرد را پیشنهاد کرد. نقاط عملکرد با استفاده از یک رابطه تجربی مبتنی بر موازین قابل شمارش (مستقیم) از دامنه اطلاعات نرم افزاری و ارزیابی پیچیدگی نرم افزار، به دست می آیند.

نقاط عملکرد با کامل کردن جدول ٤-٥ محاسبه می شوند پنج ویژگی از دامنه اطلاعاتی تعیین می شوند و شمارش هایی در محل مناسب جدول صورت می پذیرد مقادیر دامنه اطلاعات به شیوه زیر تعیین می شوند.

تعداد ورودی های کاربر. هر ورودی کاربر که داده های کاربردی متمایز نسبت به نرم افزار داشته باشد، شمارش می شوند. ورودی ها باید از درخواست ها، که جداگانه شمارش می شوند، متمایز شوند.

تعداد خروجی های کاربر. هر خروجی کاربر که اطلاعاتی با جهت گیری کاربردی برای کاربر فراهم آورد، شمارش می شود، در این معنا خروجی به گزارش ها، صفحات نمایشی، پیام های خطا و غیره اطلاق می شود. عناصر داده ای انفرادی در متن یک گزارش، به طور جدگانه شمارش نمی شوند.

تعداد درخواست های کاربر. درخواست به عنوان یک ورودی on-line تعریف می شود که منجر به تولید پاسخ بی درنگ به شکل خروجی on-line می شود هر یک از درخواست ها شمارش می شود.

تعداد فایلها. هر فایل اصلی منطقی، یعنی گروهی منطقی از داده ها که ممکن است بخشی از یک بانک اطلاعاتی بزرگ یا یک فایل مجزا باشد شمارش می شود.

تعداد واسط های خارجی. همه واسطهای خواندنی ماشین (مانند فایل های داده ای روی نوار باریک) که برای انتقال دادن اطلاعات به سیستم دیگری به کار می روند، شمارش می شوند.

FIGURE 4.5
Computing function points

Measurement parameter	Count		Weighting factor Simple	Average	Complex		
Number of user inputs	☐	×	3	4	6	=	☐
Number of user outputs	☐	×	4	5	7	=	☐
Number of user inquiries	☐	×	3	4	6	=	☐
Number of files	☐	×	7	10	15	=	☐
Number of external interfaces	☐	×	5	7	10	=	☐
Count total →							☐

شکل ٥-٤ محاسبه نقاط عملکرد

هنگامی که داده های فوق جمع آوری شد، برای هر شمارش یک مقدار پیچیدگی مشخص می شود. سازمان هایی که روش های نقاط عملکرد را به کار می برند، ملاک هایی برای تعیین این که مدخلی ساده، متوسط یا پیچیده است، توسعه می دهند با این حال، تعیین پیچیدگی قدری موضوعی است برای نقاط عملکرد (FP) از رابطه زیر استفاده شود.

$$FP=\text{شمارش کل}*[0/65+0/01*\Sigma(F_i)]$$

که در آن، شمارش کل، حاصل جمع همه مدخل های FP است که از شکل ٥-٤ به دست آمده است.

F_i ها (i=١ تا ١٤) «مقادیر تنظیم پیچیدگی» مبتنی بر پاسخ هایی است که به پرسش های زیر داده می شوند:

١. آیا سیستم به پشتیبانی و بازیابی قابل اطمینان نیاز دارد؟
٢. آیا ارتباطات داده ای مورد نیاز است؟
٣. آیا عملیات پردازشی توزیع شده وجود دارند؟
٤. آیا کارآیی اهمیت دارد؟
٥. آیا سیستم در یک محیط عملیاتی موجود و پرکاربرد اجرا می شود؟

۶. آیا سیستم به مدخل داده های on-line نیاز دارد؟
۷. آیا مدخل داده های on-line نیاز به ساخت تراکنش ورودی روی عملیات یا صفحات نمایش چندگانه دارد؟
۸. آیا فایل های اصلی به صورت on-line به هنگام می شوند؟
۹. آیا ورودی ها خروجی ها، فایل ها و در خواست های پیچیده اند؟
۱۰. آیا پردازش داخلی پیچیده است؟
۱۱. آیا کد طراحی شده است که دوباره قابل استفاده باشد؟
۱۲. تبدیل و نصب در طراحی لحاظ شده است؟
۱۳. آیا سیستم برای نصب چندگانه در سازمان های متفاوت طراحی شده است؟
۱۴. آیا برنامه کاربردی طوری طراحی شده است که تغییرات را تسهیل کند و به آسانی توسط کاربر استفاده شود؟

با استفاده از مقیاسی که از صفر (عدم اهمیت یا لزوم اجرا) تا ۵ (مطلقا ضروری) تغییر می کند، به هر یک از این پرسش ها پاسخ داده می شود مقادیر ثابت در معادله (۱-٤) و ضرایب وزنی که در شمارش دامنه اطلاعاتی به کاربرده شده اند، به طور تجربی به دست آمده اند.

هنگامی که نقاط عملکرد محاسبه شدند، از آنها به شیوه ای مشابه LOC، برای نرمال سازی موازین مربوط به بهره وری نرم افزاری، کیفیت و صفات دیگر استفاده می شود.

٤-۳-۳ معیارهای نقاط عملکرد گسترش یافته

میزان نقطه عملکرد، در ابتدا برای استفاده در کاربردهای سیستم اطلاعات تجاری طراحی شد. برای پاسخ‌گویی به این نیازها، بعد داده ها (مقادیر دامنه اطلاعاتی بحث شده در بالا) از ابعاد عملیاتی و رفتاری (کنترلی) جدا شد. به این دلیل میزان نقاط عملکرد برای بسیاری از سیستم های مهندسی و ادغام شده (که بر عملکرد و کنترل تاکید دارند) نارسا بود، برای رفع این نقیصه مواردی پیشنهاد گردیده است.

شکل گسترش یافته نقطه عملکرد که نقطه ویژگی نام دارد. اَبر مجموعه ای از میزان نقاط عملکرد است که در کاربردهای مهندسی نرم افزار و سیستم ها قابل اجرا هستند نقطه ویژگی، کاربردهایی را شامل می شود که در آن پیچیدگی الگوریتمی بالاست، در کاربردهای بلادرنگ، کنترل فرآیند و نرم افزارهای تعبیه شده، پیچیدگی الگوریتم ها بالاست و در نتیجه در نقطه ویژگی می گنجد.

برای محاسبه نقطه ویژگی، مقادیر دامنه اطلاعاتی دوباره شمارش می شوند و به صورت شرح داده شده در بخش قبل توزین می شود به علاوه معیار، نقطه ویژگی، یک ویژگی نرم افزار جدید الگوریتم ها را شمارش می کند. یک الگوریتم به عنوان یک مسئله محاسباتی مقید تعریف می شود که در یک برنامه کامپیوتری مشخص گنجانده می شود. معکوس کردن یک ماتریس، رمزگردانی یک رشته بیتی، یا کار با یک وقفه، همگی مثال هایی از الگوریتم هستند.

یک شکل گسترش یافته دیگر برای سیستم های بلادرنگ و محصولات مهندسی شده، توسط بوئینگ ابداع شد در روش بوئینگ، بعد داده ای نرم افزار با ابعاد عملیاتی و کنترلی گرد هم آورده شدند تا یک میزان عملکردگرا برای کاربردهایی که بر قابلیت های عملیاتی و کنترلی تاکید دارند، به دست آید، در یک میزان که نقطه عملکرد سه بعدی نامیده می شود و قابلیت عملیاتی تحویل شده توسط نرم افزار را نشان می دهد، خصوصیات هر سه بعد نرم افزار شمارش، تعیین مقدار و تبدیل می شوند.

بعد داده ای تا حد زیادی به همان صورت شرح داده شده در بخش قبل ارزیابی می شود تعداد داده های حفظ شده (ساختمان داده های برنامه داخلی، مانند فایل ها) و داده های خارجی (ورودی ها، خروجی ها، درخواست ها و مراجع خارجی) همراه با موازینی از پیچیدگی مورد استفاده قرار می گیرند تا شمارشی از بعد داده ای به دست آید. بعد عملیاتی با در نظر گرفتن تعداد اعمال داخلی لازم برای تبدیل داده های ورودی به خروجی اندازه گیری می شود برای محاسبه نقطه عملکرد سه بعدی، تبدیل به عنوان یک سری مراحل پردازشی در نظر گرفته می شود که تحت انقیاد مجموعه ای از جمله های معنی دار است بعد کنترلی با شمارش تعداد گذارها بین حالت ها اندازه گیری می شود.

هر حالت، نشانگر یک شیوه رفتاری است که از بیرون قابل مشاهده است و گذار، نتیجه رویدادی است که باعث می شود نرم افزار یا سیستمی شیوه رفتار خود را تغییر دهد (یعنی تغییر حالت دهد) برای مثال، یک تلفن بی سیم دارای نرم افزاری است که عملیات شماره گیری خودکار را پشتیبانی می کند برای وارد شدن به حالت شماره گیری خود از حالت سکون، کاربرد کلید Auto را فشار می دهد این رویداد باعث می شود تا صفحه نمایش LCD کد مخاطب را درخواست کند. پس از وارد کردن کد و فشار کلید Dial (یک رویداد دیگر) نرم افزار تلفن بی سیم به حالت شماره گیری گذار می کند هنگام محاسبه نقاط عملکرد سه بعدی، به گذارها پیچیدگی نسبت داده نمی شود.

برای محاسبه نقاط عملکرد سه بعدی، از رابطه زیر استفاده می شود.

$$\text{شاخص} = I+O+Q+F+E+T+R$$

که در آن I، O، Q، F، E، T و R به ترتیب نشانگر مقادیر توزین شده پیچیدگی برای عناصر بحث شده در بالا یعنی ورودی ها، خروجی ها، درخواست ها، ساختمان داده داخلی، فایل های خارجی، تبدیلات، و تراکنش ها هستند هر مقدار توزین شده پیچیدگی با استفاده از رابطه زیر محاسبه می شود.

$$\text{مقدار توزین شده پیچیدگی} = N_{il}\,W_{il} + N_{ia}\,W_{ia} + N_{ih}\,W_{ih}$$

که N_{ih}، N_{ia}، N_{il} تعداد فراوانی عنصر i(مثلا خروجی ها) به ازای هر سطح از پیچیدگی (کم، متوسط، زیاد) را نشان می دهد W_{il}، W_{ia}، W_{ih} اوزان مرتبط با آنها هستند محاسبه کامل نقاط عملکرد سه بعدی در شکل ۶-٤ نشان داده شده است.

FIGURE 4.6 Determining the complexity of a transformation for 3D function points [WHI95].

Processing steps \ Semantic statements	1-5	6-10	11+
1-10	Low	Low	Average
11-20	Low	Average	High
21+	Average	High	High

شکل ۶-۴ محاسبه شاخص نقطه عملرد سه بعدی

باید توجه داشت که نقاط عملکرد، نقاط ویژگی و نقاط عملکرد سه بعدی همه نشانگر یک چیز هستند قابلیت عملیاتی یا کارکرد تحویل شده توسط نرم افزار. در واقع همه این موازین فقط در صورتی به یک مقدار یکسان می انجامند که بعد داده ای یک کاربرد در نظر گرفته شود، برای سیستم های بلادرنگ، شمار نقاط ویژگی غالبا ۲۰ تا ۳۵ درصد بیشتر از شمار تعیین شده توسط نقاط عملکرد است.

نقطه عملکرد (و اشکال گسترش یافته آن)؛ همانند LOC موضوعی بحث انگیز است هواداران آن ادعا می کنند که FP مستقل از زبان برنامه نویسی بوده لذا برای کاربردهایی که از زبانهای قراردادی و غیر رویه ای استفاده می کنند ایده آل است، یعنی مبتنی بر داده هایی است که به احتمال زیاد در همان مراحل ابتدایی تکامل پروژه معلوم می شوند و از این رو FP، روشی برای برآورد است مخالفان ادعا می کنند که این روش نیاز به قدری چیره دستی دارد زیرا محاسبه در آن مبتنی بر داده های موضوعی است نه داده های عینی، جمع آوری شمار دامنه های اطلاعاتی (و ابعاد دیگر) می تواند دشوار باشد و FP فاقد هر گونه معنای فیزیکی بوده صرفا یک عدد است.

٤-٤ ایجاد ارتباط میان معیارهای متفاوت

رابطه میان خطوط کد و نقاط عملکرد، به کیفیت طراحی و زبان برنامه سازی بستگی دارد که برای پیاده سازی نرم افزار، مورد استفاده قرار گرفته است در چندکار پژوهشی، کوشش شده تا بین دو میزانFP و LOC ارتباطی برقرار شود، آلبرشت و گافنی می گویند:

غرض از این کار پژوهشی آن است که مقدار عملکرد ارائه شده توسط برنامه کاربردی را بتوان از روی تک تک قطعات اصلی داده های مورد استفاده آن یا تولید شده توسط آن برآورد نمود. به علاوه این برآورد، عملکرد باید هم با LOC تولید شده و هم انرژی صرف شده همبستگی داشته باشد.

جدول زیر برآورد خاصی از میانگین تعداد خطوط کد مورد نیاز برای ساخت یک نقطه عملکرد را در زبان های برنامه نویسی گوناگون نشان می دهد:

? If I know the number of LOC, is it possible to estimate the number of function points?

Programming Language	LOC/FP (average)
Assembly language	320
C	128
COBOL	106
FORTRAN	106
Pascal	90
C++	64
Ada95	53
Visual Basic	32
Smalltalk	22
Powerbuilder (code generator)	16
SQL	12

مرور داده های فوق نشان می دهد که قابلیت عملیاتی یک LOC از C++ حدود ۱/۶ برابر یک LOC از فرترن است به علاوه قابلیت عملیاتی یک LOC از ویژوال بیسیک بیش از سه برابر قابلیت عملیاتی یک زبان برنامه نویسی مرسوم

backfire کردن برنامه های موجود در تعیین FP برای هر یک از آنها استفاده کرد (منظور از backfire کردن محاسبه تعداد نقاط عملکرد در هنگامی است که تعداد LOC تحویل شده مشخص باشد).
از میزان های LOC و FP غالبا برای به دست آوردن معیارهای بهره وری استفاده می شود این خود ناگزیر منجر به بحث درباره استفاده از این گونه داده ها می شود آیا LOC به ازای هر نفر – ماه (یا FP به ازای هر نفر ماه) در یک گروه را باید با گروه دیگر مقایسه کرد؟ آیا مدیران باید کارآیی افراد را با استفاده از این معیارها ارزیابی کنند؟ پاسخ این پرسش ها یک نه محکم است دلیل این پاسخ این است که بسیاری از عوامل بهره وری را تحت تاثیر قرار می دهند، به طوری که ممکن است سیب و پرتقال با هم اشتباه گرفته شوند.
معلوم شده است که معیارهای مبتنی بر LOC و نقاط عملکرد برای پیش بینی هزینه و انرژی لازم برای توسعه نرم افزار، نسبتا صحیح عمل می کنند ولی برای استفاده از LOC و FP در تکنیک های تخمین زنی باید سابقه ای از اطلاعات در دسترس باشد.

٤-٥ معیارهای مربوط به کیفیت نرم افزار

هدف اصلی مهندسی نرم افزار، تولید سیستم، برنامه کاربردی، یا محصولی با کیفیت بالاست، برای نیل به این هدف، مهندسان نرم افزار باید روش های کارآمدی را همراه با ابزارهای مدرن در بطن یک فرآیند نرم افزاری بلوغ یافته به کار بندند. به علاوه یک مهندس نرم افزار خوب (و مدیران نرم افزار خوب) اگر به کیفیت بالا اهمیت می دهند باید اندازه گیری کنند.
کیفیت یک سیستم برنامه کاربردی یا محصول، از حد خواسته هایی که مسئله را توصیف می کنند طراحی که راهکار را مدل سازی می کند، کدی که منجر به برنامه اجرایی می شود و آزمایش هایی که طی آن خطاهای نرم افزار آشکار می گردد، فراتر نمی رود، یک مهندس نرم افزار خوب برای ارزیابی کیفیت تحلیل و مدل های طراحی، کد منبع و موارد آزمایشی را باید اندازه گیری کند که به موازات مهندسی شدن نرم افزار ایجاد شده اند برای دستیابی به این ارزیابی کیفیت در بلادرنگ، مهندس باید از موازین فنی برای ارزیابی کیفیت به شیوه ای عمیق و نه موضوعی استفاده کند.
مدیر پروژه باید کیفیت را نیز به موازات پیشرفت پروژه مورد ارزیابی قرار دهد، برای فراهم آوردن نتایجی در سطح پروژه از معیارهای خصوصی جمع آوری شده توسط تک تک مهندسان نرم افزار استفاده می شود گرچه بسیاری از موازین کیفیتی را می توان جمع آوری کرد، گام اول در سطح پروژه، اندازه گیری خطاها و نقایص است معیارهای به دست آمده از این موازین، شاخصی از کارآمد بودن تضمین کیفیت گروهی، و فردی و فعالیت های کنترلی فراهم می آوردند.

٤-٥-١ بررسی اجمالی عوامل موثر بر کیفیت

بالغ بر ۲۵ سال قبل، مک کال و کارانو مجموعه ای از عوامل کیفیتی را تعریف کردند که نخستین گام به سوی توسعه معیارهایی برای کیفیت نرم افزار بود این عوامل را از سه دیدگاه ارزیابی می کنند. ۱. کارکرد محصول (استفاده از آن) ۲. بازبینی محصول (تغییر دادن آن) و ۳. گذار محصول (اصلاح آن برای کارکردن در محیطی متفاوت). این پژوهشگران در کار خود رابطه میان این عوامل کیفیتی (که آن را چارچوب می نامند) و جنبه های دیگر فرآیند مهندسی نرم افزار را چنین شرح می دهند:

اولا چارچوب، راهکاری برای مدیر پروژه فراهم می آوردکه بداند کدام کیفیت ها اهمیت دارند. این کیفیتها صفاتی از نرم افزار علاوه بر صحت عملیاتی و کارآیی است که چرخه حیات آن را تشکیل می دهند. طی سالها نشان داده شده است که عواملی از قبیل قابلیت نگهداری و حمل پذیری، تاثیر چشمگیری بر هزینه و چرخه حیات نرم افزار داشته اند.

ثانیا چارچوب، ابزاری برای ارزیابی کمی چگونگی پیشرفت توسعه به لحاظ اهداف کیفیتی وضع شده فراهم می آورد.

ثالثا چارچوب، تعامل بیشتر پرسنل QA(تضمین کیفیت) در سرتاسر توسعه را فراهم می آورد.

و بالاخره، پرسنل QA می توانند از شاخص های کیفیت ضعیف برای کمک به شناخت (بهتر) استانداردهایی که در آینده تحمیل خواهد شد استفاده کنند.

جالب است متذکر شویم که تقریبا هر جنبه از علم کامپیوتر، از آن زمان که مک کال و کاوانو کار خود را در سال ۱۹۷۸ ارائه دادند، دستخوش تغییرات بنیادی شده است ولی صفاتی که ارائه دادند، دستخوش تغییرات بنیادی شده است اما صفاتی که شاخص کیفیت نرم افزار هستند، همچنان ثابت باقی مانده اند.

این چه معنایی دارد؟ اگر یک سازمان نرم افزاری مجموعه ای از عوامل کیفیتی را به عنوان لیست کنترلی برای ارزیابی کیفیت نرم افزاها مدنظر قرار دهد احتمال آن وجود دارد که نرم افزاری که امروز ساخته شده است در اولین دهه های قرن حاضر نیز کیفیت خروجی از خود نشان دهد. حتی اگرمعماری های کامپیوتر دستخوش تغییرات بنیادی بشوند (که مطمئنا می شوند) نرم افزاری که در راه اندازی، گذار و بازبینی، کیفیت خوبی از خود نشان می دهد همچنان در خدمت کاربران خود باقی می ماند.

۲-۵-٤ اندازه گیری کیفیت

گرچه موازین فراوانی برای کیفیت نرم افزار وجود دارند، درستی، قابلیت نگهداری، یکپارچگی و قابلیت استفاده، شاخص های مفیدی برای تیم پروژه فراهم می آورند، گلیب برای هر یک تعاریف و موازینی را پیشنهاد کرده است.

درستی.یک برنامه باید درست کار کند و گرنه برای کاربران خود ارزشی ندارد درستی، حدی از کارکرد نرم افزار برای انجام وظایف آن است، متداولترین میزان برای درستی، تعداد نقایص به ازای هر KLOC است منظور از نقص، عدم مطابقت تصدیق شده با خواسته هاست.

قابلیت نگهداری. نگهداری نرم افزار از هر فعالیت مهندسی نرم افزار دیگر بیشتر کار می برد، قابلیت نگهداری عبارت از سهولت تصحیح یک برنامه در صورت مواجهه با خطا، تطابق با تغییرات محیط یا بهبود بخشیدن به برنامه در صورت تغییر یافتن خواسته های مشتری است راهی برای اندازه گیری مستیم قابلیت نگهداری وجود ندارد، بنابراین، باید از موازین غیر مستقیم استفاده شود یک معیار مبتنی بر زمان میانگین زمان لازم برای تغییر (MTTC) است، یعنی زمان لازم برای تحلیل تقاضای تغییر، طراحی یک اصلاح مناسب، پیاده سازی تغییر، آزمون آن و توزیع تغییر در میان همه کاربران. به طور میانگین، برنامه هایی که قابل نگهداری هستند از برنامه هایی که قابل نگهداری نیستند MTTC کمتری دارند.

هیتاپی یک معیار مبتنی بر هزینه را برای قابلیت نگهداری به کار برده است که ریخت و پاش نام دارد و عبارت از هزینه ای است که برای رفع نواقصی صرف می شود که پس از ارائه سیستم به کاربران شناسایی شدند هنگامی که نسبت ریخت و پاش به هزینه کل پروژه (برای چند پروژه) به صورت تابعی از زمان رسم شود، مدیر می تواند تعیین کند که آیا قابلیت نگهداری کلی نرم افزار تولید شده، توسط سازمان بهبود یافته است یا خیر. سپس می توان با دیدی که از این اطلاعات به دست می آید عمل مناسب را اتخاذ نمود.

یکپارچگی. یکپارچگی نرم افزار با توجه به نفوذگرها و حفاظ ها اهمیتی روز افزون ساخته است این صفت، میزانی از توانایی سیستم برای تحمل حملاتی (چه تصادفی چه عمدی) است که متوجه امنیت آن می شود این حملات روی هر سه مولفه نرم افزار صورت می پذیرد، برنامه ها، داده و مستندات.
برای اندازه گیری یکپارچگی، دوصفت دیگر را باید تعیین کرد، تهدید و امنیت. تهدیدی عبارت از احتمال وقوع حمله ای از یک نوع خاص در زمانی معین است (که می توان آن را از روی شواهد تجربی تخمین زد یا به دست آورد). امنیت احتمال دفع حمله ای از یک نوع خاص است (که آن را نیز از روی شواهد تجربی می توان تخمین زد یا به دست آورد) پس یکپارچگی سیستم را چنین می توان تعریف کرد:

یکپارچگی =Σ[(امنیت-۱) * (تهدید -۱)]

که در آن تهدید و امنیت روی همه انواع حملات جمع بسته می شوند.
قابلیت استفاده. عبارت سهولت استفاده در بحث های محصولات نرم افزاری به وفور مشاهده می شود اگر استفاده از برنامه ای آسان نباشد سرنوشت آن شکست خواهد بود، حتی اگر اعمالی که انجام می دهد، ارزشمند باشد قابلیت استفاده کوششی برای تعیین کمی سهولت استفاده برحسب چهار خصوصیت قابل اندازه گیری است ۱) مهارت فیزیکی و هوشی لازم برای فراگیری سیستم۲) زمان لازم برای بازدهی خوب در به کارگیری سیستم ۳) افزایش خالص در بهره وری (نسبت به روشی که جایگزین می شود) و ٤) یک ارزیابی موضوعی از احساس کاربران نسبت به سیستم (که غالبا از طریق پرسشنامه به دست می آید) چهار عاملی که در بالا بحث شد تنها نمونه از موارد پیشنهاد شده به عنوان موازینی از کیفیت نرم افزار هستند.

۳-۵-٤ بازدهی رفع نقص

یک معیار کیفیتی که هم در سطح پروژه و هم در سطح فرآیند مفید واقع می شود، معیار بازدهی رفع نقص است در اصل DRE، میزانی از توانایی فیلتر کردن فعالیت های مربوط به کنترل و تضمین کیفیت است که در سرتاسر فعالیت های چارچوبی اجرا می شوند.
هنگامی که DRE برای کل پروژه در نظر گرفته می شود به شیوه زیر تعریف می شود.

$$DRE=E/(E+D)$$

که در آن E تعداد خطاهای یافت شده، پیش از تحویل نرم افزار به کاربر نهایی و D تعداد نقایص یافت شده پس از تحویل است مقدار ایده آل برای DRE برابر با یک است یعنی این که هیچ نقصی در نرم افزار یافت نشود در حالت واقعی، D بزرگ تر از صفر است ولی مقدار DRE باز هم می تواند به یک نزدیک شود با افزایش E برای مقدار معلومی از D مقدار DRE به یک نزدیک می شود در واقع با افزایش E، احتمال کاهش مقدار نهایی D بیشتر می شود (خطاها پیش از آنکه به نقص تبدیل شوند فیلتر می شوند) اگر DRE به عنوان معیاری به کار گرفته شود که شاخصی از توانایی فیلتر کردن فعالیت های کنترل و تضمین کیفیت را به دست دهد تیم نرم افزاری را به وضع تکنیک هایی تشویق می کند که خطاها را پیش از تحویل سیستم به کاربران نهایی بیابند.

DRE را در داخل خود پروژه نیز می توان برای ارزیابی توانایی تیم، در یافتن خطاها پیش از تحویل به فعالیت چارچوبی بعدی یا وظیفه مهندسی نرم افزاری بعدی بکار برد. برای مثال وظیفه تحلیل خواسته ها، یک مدل تحلیل ایجاد می کند که می توان برای یافتن و تصحیح خطاها از آن استفاده کرد.
آن دسته از خطاها که طی بازبینی مدل تحلیل یافت نمی شوند به وظیفه طراحی راه پیدا می کنند (که در آنجا ممکن است یافته شوند یا نشوند) در این حالت DRE به صورت زیر تعریف می شود.

$$DRE_i = E_i / (E_i + E_{i+1})$$

که در آن E_i تعداد خطاهای یافت شده طی فعالیت مهندسی نرم افزار i و E_i+1 تعدا خطاهای یافت شده طی فعالیت مهندسی نرم افزار i+1 است که در فعالیت مهندسی نرم افزار i کشف نشده اند.

فصل ۱۳: اصول و مفاهیم طراحی

طراحی چیست؟ طراحی نمایش مهندسی معناداری از چیزی است که باید ساخته شود. باید بر اساس خواسته های کاربر و ضوابط موجود ردیابی شود. در زمینه مهندسی نرم افزار طراحی به چهار مورد می پردازد: داده ها، معماری، واسط ها و مؤلفه ها. مفاهیم وقواعدی که در این فصل بحث می شود، به هر چهار مورد اعمال می گردد.

چه کسی طراحی را انجام می دهد؟ مهندسین نرم افزار، **سیستم های** کامپیوتری را طراحی می کنند ولی مهارت های مورد نیاز در هر سطح متفاوت است. در سطوح داده ها و معماری، طراحی بر روی الگو هایی تأکید دارد که به برنامه کاربردی که باید ساخته شود اعمال می شوند. در سطح واسط مسائل ارگونومیک حرف اول را می زنند. در سطح مؤلفه، «روش برنامه نویسی» ما را به طرح های مؤثر داده و رویه ها هدایت می کند.

چرا طراحی مهم است؟ هیچ کس خانه ای را بدون نقشه نمی سازد. نرم افزار کامپیوتر به مراتب پیچیده تر از ساختمان است بنابرین نیاز به نقشه است(طراحی).

مراحل طراحی چیست؟ طراحی با مدل خواسته ها شروع می شود. این مدل را به چهار سطح طراحی تبدیل می کنیم : ساختمان داده، معماری سیستم، نمایش واسط، وجزئیات سطح مؤلفه. در اثنای هر فعالیت طراحی از مفاهیم و قواعدی استفاده می کنیم تا کیفیت کار بالا باشد.

محصول کار چیست؟ معمولا یک مشخصات طراحی ایجاد می شود. این مشخصات حاوی مدل های طراحی است که داده ها، معماری، واسط ها و مؤلفه ها را توصیف می کند، هر کدام از این ها یک محصول طراحی است.

چگونه مطمئن شویم که طراحی به درستی انجام شد؟ در هر مرحله، محصولات طراحی نرم افزار مرور می شوند تا وضوح، صحت، تمامیت وسازگاری آنها درست باشد.

۱۳-۱ طراحی نرم افزار و مهندسی نرم افزار

طراحی نرم افزار هستهٔ فنی مهندسی نرم افزار را تشکیل می دهد و مستقل از مدل فرآیند نرم افزار اعمال می شود. هنگامی که خواسته های نرم افزار تحلیل و مشخص شد، شروع طراحی نرم افزار نخستین فعالیت فنی از سه فعالیت – طراحی، تولید کد و آزمایش است که برای ساخت و بازرسی نرم افزار مورد نیاز هستند. هر یک از این فعالیت ها، اطلاعات را به شیوه ای تبدیل می کند که در نهایت به نرم افزاری معتبر ختم شود.

هر یک از عناصر مدل تحلیل اطلاعاتی فراهم می آورد که برای ایجاد چهار مدل طراحی موردنیاز در تعیین مشخصات کامل نرم افزار لازمند. جریان اطلاعات در اثنای طراحی نرم افزار در شکل ۱-۱۳ نشان داده شده است.

خواسته های نرم افزار که توسط مدل های داده ای عملیاتی و رفتاری اعلام می شوند وظیفهٔ طراحی را تغذیه می کنند.

وظیفهٔ طراحی با به کارگیری یکی از چند روش طراحی یک طراحی داده ها، یک طراحی معماری، یک طراحی واسط و یک طراحی مؤلفه ها را ایجاد می کند.

طراحی داده ها

مدل دامنه اطلاعاتی ایجاد شده در اثنای تحلیل را به ساختمان داده ای تبدیل می کند که برای پیاده سازی نرم افزار لازم هستند. اشیای داده ای و روابط تعریف شده در نمودار نهاد- رابطه و محتویات داده ها که در دیکشنری داده ها وارد شده اند مبنایی برای فعالیت طراحی داده ها فراهم می آورند. بخشی از طراحی داده ها ممکن است همراه با طراحی معماری صورت پذیرد. طراحی مفصلتر داده ها به موازات طراحی هر یک از مؤلفه های نرم افزار صورت می پذیرد.

طراحی معماری، رابطهٔ میان عناصر ساختاری نرم افزار، الگو های طراحی که می توان آنها را برای برآوردن خواسته های تعریف شده برای سیستم به کاربرد، و نیز محدودیت هایی که شیوهٔ به کارگیری الگو های طراحی معماری را تحت تأثیر قرار می دهند، تعیین می کند. نمایش طراحی معماری – چارچوب سیستم کامپیوتری – را می توان از روی مشخصات سیستم، مدل تحلیل، و تأثیر متقابل زیرسیستم های تعریف شده در مدل تحلیل به دست آورد.

طراحی واسط، چگونگی برقراری ارتباط داخلی سیستم، ارتباط آن با سیستم های وابسته به آن و انسانهایی را که با آن کار می کنند، توصیف می کند. یک واسط، حاکی از جریان اطلاعات (مثلا داده ها و/ یا کنترل)و نوع خاصی از رفتار است از اینرو، نمودارهای جریان داده ها و کنترل، اکثر اطلاعات لازم برای طراحی واسط را فراهم می آورند.

طراحی در سطح مؤلفه ها، عناصر ساختاری معماری نرم افزار را به یک توصیف رویه ای از مؤلفه های نرم افزار تبدیل می کند. اطلاعات به دست آمده از STD ,CSPEC, PSPEC مبنای طراحی مؤلفه ها را تشکیل می دهند.

اهمیت طراحی نرم افزار را در یک کلام می توان خلاصه کرد : کیفیت طراحی نقطه ای است که در آن کیفیت وارد مهندسی نرم افزار می شود. طراحی، نمایشی از نرم افزار را در اختیار قرار می دهد که می توان آن را از لحاظ کیفیت مورد ارزیابی قرار داد.

طراحی، تنها راهی است که خواسته های مشتری را به سیستم یا محصول نرم افزاری تبدیل می کند. طراحی نرم افزار به عنوان مبنایی برای تمام مراحل مهندسی نرم افزار و پشتیبانی نرم افزار عمل می کند. بدون طراحی، خطر ساخت بنایی ناپایدار وجود دارد – یعنی بنایی که با مختصر تغییراتی فرو می ریزد؛ بنایی که آزمایش آن دشوار است و بنایی که کیفیت آن را نمی توان ارزیابی کرد مگر در واپسین قدمهای فرآیند نرم افزار، یعنی جایی که زمان کوتاه است و مبالغ هنگفتی هزینه شده است.

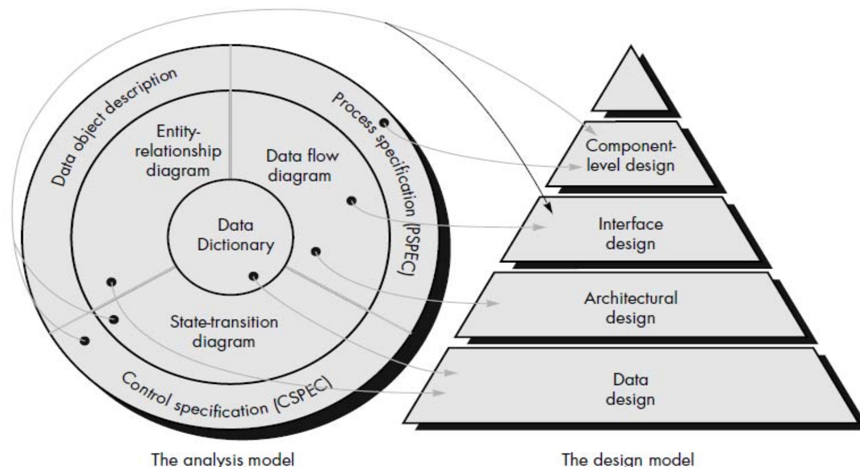


FIGURE 13.1 Translating the analysis model into a software design

شکل ۱-۱۳ تبدیل تحلیل به طراحی نرم افزار

۲- ۱۳ فرآیند طراحی

طراحی نرم افزار، یک فرآیند تکراری است که از طریق آن خواسته ها به نقشه ای برای ایجاد کردن نرم افزار ترجمه می شوند. این نقشه در آغاز، نمایی کلی از نرم افزار به تصویر می کشد یعنی نرم افزار در سطح بالایی از انتزاع نمایش داده می شود – سطحی که به طور مستقیم تا اهداف مشخص سیستم، خواسته های داده ای تفضیلی، خواسته های عملیاتی و خواسته های رفتار قابل پیشگیری باشد. به موازاتی که تکرار ها رخ می دهد، بهسازیهای بعدی منجر به نمایش هایی از طراحی در سطوح پایین تر انتزاع می شوند. این سطوح نیز تا خواسته ها قابل پیگیری هستند ولی ارتباط میان آنها ظریفتر است.

۱۳-۲-۱ طراحی و کیفیت نرم افزار

در سرتاسر فرآیند طراحی، کیفیت طراحی در حال تکامل، با یک سری بازبینی های فنی رسمی یا مرورهای مقدماتی طراحی مورد ارزیابی قرار می گیرد. مک گلوگلین سه ویژگی را پیشنهاد می کند که به عنوان راهنمایی برای ارزیابی یک طراحی خوب عمل می کنند:

- طراحی باید کلیه خواسته های واضح موجود در مدل تحلیل را پیاده سازی کرده، جایی برای تمامی خواسته های مبهم مشتری در نظر بگیرد؛
- طراحی باید برای کسانی که کد را تولید می کنند، کسانی که نرم افزار را آزمایش می کنند و کسانی که آن را پشتیبانی می کنند راهنمایی قابل فهم باشد.
- طراحی باید تصویر کاملی از نرم افزار ارائه دهد که دربرگیرنده دامنه های رفتاری، عملیاتی و داده ای از دیدگاه پیاده سازی باشد.

جهت ارزیابی کیفیت نمایش طراحی، باید ملاکهای فنی برای طراحی خوب وضع نمود. بعدا در همین فصل ملاک های طراحی خوب را به تفصیل مورد بحث قرار خواهیم داد. فعلا دستورالعمل های زیر را ارائه می دهیم:

۱- طراحی باید ساختار معماریی داشته باشد که a. با استفاده از الگوهای طراحی شناخته شده ایجاد شده باشد؛b. از مؤلفه هایی تشکیل شده باشد که ویژگی های طراحی خوبی از خود نشان می دهند c. به شیوه ای تکاملی قابل پیاده سازی بوده در نتیجه پیاده سازی و آزمون آن آسان باشد.

۲- طراحی باید پیمانه ای باشد؛ یعنی نرم افزار باید به طور منطقی به عناصری افراز شده باشد که عملیات خاصی را اجرا می کنند.

۳- طراحی باید حاوی نمایش برجسته ای از داده ها، معماری، واسط ها و مؤلفه ها (پیمانه ها) باشد.

٤- طراحی باید منجر به ساختمان داده ای شود که مناسب اشیایی باشد که قرار است پیاده سازی شوند و از الگوهای داده ای قابل تشخیص گرفته شود.

۵- طراحی باید منجر به مؤلفه هایی شود که ویژگی های عملیاتی مستقلی از خود نشان دهند.

۶- طراحی باید به واسطهایی منجر شود که پیچیدگی ارتباطات میان پیمانه ها و محیط خارجی را کاهش دهد.

۷- طراحی باید با استفاده از یک روش تکرار به دست آید که توسط اطلاعات حاصل از تحلیل خواسته های نرم افزار هدایت می شوند.

این ملاکها به طور اتفاقی به دست نمی آیند. فرآیند طراحی نرم افزار، از طریق بکار گیری اصول طراحی بنیادی، روش شناسی سیستماتیک و باز بینی کامل، طراحی خوب را تشویق می کند.

۱۳-۲-۲ تکامل طراحی نرم افزار

طراحی نرم افزار هم یک فرآیند و هم یک مدل است. فرآیند طراحی عبارت از یک سری مراحلی است که طراح را قادر به توصیف تمامی جنبه های نرم افزاری می سازد که قرار است ساخته شود. ولی توجه به این نکته حائز اهمیت است که فرآیند طراحی، صرفا یک کتاب آشپزی نیست. مهارتهای خلاقانه، تجربیات پیشین، برداشتی از یک نرم افزار خوب وکوشش کلی برای دستیابی به کیفیت از عوامل موفقیت یک طراح به شمار می رود.

مدل طراحی همتای نقشه های معماری ساختمان است. مدل با کلیت چیزی که قرار است ساخته شود (مثلا پرداخت سه بعدی ساختمان) آغاز شده رفته رفته آن را پالایش می کند تا راهنمایی برای ساخت هر یک از جزئیات (مثلا آرایش لوله کشی) فراهم آید. به طور مشابه، مدل طراحی برای نرم افزار، چند نمای متفاوت از نرم افزار کامپیوتری به دست می دهد.

۱۳-۳ اصول طراحی پایه

مهندس نرم افزار را قادر می سازد تا در فرآیند طراحی به تفحص بپردازد. دیویس یک مجموعه اصول برای طراحی نرم افزار پیشنهاد می کند که آنها را در لیست زیر تنظیم کردیم:

- در فرآیند طراحی نباید دیدی تک بعدی داشت. طراح خوب باید روشهای متفاوت را در نظر بگیرد و بر اساس خواسته های مسئله، منابع در دسترس و مفاهیم ارائه شده در بخش ٤-١٣، هر یک از آنها را مورد قضاوت قرار دهد.
- مدل تحلیل باید در طراحی قابل پیگیری باشد. از آنجا که در یک عنصر از مدل طراحی غالبا جای پای چند خواسته مشاهده می شود، لازم است ابزاری برای پیگیری چگونگی رعایت خواسته ها در مدل طراحی موجود باشد.
- طراحی نباید دوباره کاری باشد. سیستم ها با استفاده از مجموعه ای از الگوهای طراحی ساخته می شوند که اکثر آنها احتمالا قبلا ابداع شده اند. زمان کوتاه و منابع محدود است! زمان طراحی باید مصروف ارائه ایده های واقعا جدید و الحاق آنها به ایده های موجود شود.
- طراحی باید «فاصله هوشی» میان نرم افزار و مسئله واقعی را به حداقل برساند. یعنی ساختار طراحی نرم افزار باید (هر گاه که امکان داشته باشد)تقلیدی از دامنۀ مسئله باشد.
- طراحی باید یکنواخت و منسجم باشد. طراحی در صورتی یکنواخت است که به نظر برسد یک نفر کل آن را انجام داده است.
- ساختار طراحی باید چنان باشد که پذیرای تغییرات باشد. مفاهیم طراحی که در بخش بعد مورد بحث قرار خواهند گرفت، طراحی را در این مورد رهنمون خواهند شد.
- ساختار طراحی باید چنان باشد که بصورت ملایم تنزل پیدا کند حتی هنگامی که به شرایط عملیاتی، رویدادها یا داده های غیر عادی برمی خورد. نرم افزاری که از طراحی خوب برخوردار باشد هرگز نباید «منفجر» شود بلکه باید طوری طراحی گردد که شرایط غیرعادی را پذیرا باشد و اگر به پردازش پایان می دهد این کار را به شیوه ای آرام انجام دهد.
- طراحی کد نویسی نیست و کد نویسی طراحی نیست. حتی هنگامی که طراحی مشروح روالها برای قطعات برنامه به پایان رسید، سطح انتزاع مدل طراحی باز هم بالاتر از کد منبع است. فقط تصمیمات طراحی اتخاذ

شده در سطح کد نویسی به جزئیات پیاده سازی مربوط می شود که کد نویسی طراحی روالها را میسر می سازد.

- سنجش کیفیت طراحی باید به موازات ایجاد آن انجام شود نه پس از پایان یافتن آن. چند مفهوم طراحی و موازین طراحی برای کمک به سنجش کیفیت طراحی در دسترس است.
- طراحی باید مورد بازبینی قرار گیرد تا خطاهای مفهومی به حداقل برسد. گاهی به هنگام بازبینی طراحی، توجه، معطوف جزئیات می شود و مثل این است که خود جنگل تحت الشعاع درخت ها قرار گیرد. یک تیم طراحی باید اطمینان یابد که عناصر مفهومی اصلی طراحی (کمبودها، ابهامات، ناسازگاری ها) پیش از پرداختن به نحو مدل طراحی، مورد توجه قرار گرفته اند

۱۳-٤-۲ پالایش

پالایش مرحله ای، یک راهبرد طراحی بالا به پایین است که اولین بار نیکلاوس ویرث آن را پیشنهاد کرد. برنامه با پالایش پیاپی سطوح جزئیات رویه ای توسعه می یابد. با تجزیهٔ بیان ماکروسکوپی عملکرد (یک انتزاع رویه ای) به شیوه ای مرحله ای، سلسله مراتبی تشکیل می شود تا به دستوراتی به زبان برنامه نویسی برسیم. ویرث دیدی کلی از این مفهوم ارائه داده است:

در هر مرحله یک یا چند دستورالعمل از برنامه مفروض به چند دستورالعمل جزئی تر تجزیه می شوند. این تجزیه یا پالایش پیاپی مشخصات زمان پایان می یابد که همهٔ دستورالعمل ها بر حسب یک زبان برنامه نویسی بیان می شوند.

به موازات پالایش وظایف داده ها نیز ممکن است پالایش یابند، تجزیه شوند یا ساختار آن ها تغییر کند و طبیعتا برنامه و مشخصات داده ها نیز به فراخور مورد پالایش قرار می گیرند.

هر مرحله پالایش مستلزم یک سری تصمیم گیریی های طراحی است. این نکته حائز اهمیت است که برنامه نویس باید از ملاک های زیربنایی (برای تصمیم گیری های طراحی) و از وجود راه کارهای دیگر آگاه باشد.

این فرآیند پالایش برنامه که ویرث پیشنهاد کرده است مشابه پالایش و تقسیمی است که طی تحلیل خواسته ها به کار می رود. اختلاف میان آنها در سطح جزئیات است نه در روش انجام کار.

پالایش در واقع یک فرآیند از هم پاشی است. کار یک بیان از عملکرد (یا توصیفی از اطلاعات) آغاز می شود که در سطح بالایی از انتزاع تعریف می شود.

یعنی این بیان عملکرد یا اطلاعات را به لحاظ مفهومی توصیف می کند، ولی اطلاعاتی در خصوص کارهای داخلی عملکرد یا ساختار داخلی اطلاعات ارائه نمی کند. پالایش سبب می شود تا طراح بیان اولیه را متلاشی کند و به ازای هر بار پالایش، جزئیات بیشتری را فراهم آورد.

۱۳-٤-۳ پیمانه ای کردن

قریب به پنج دهه است که مفهوم پیمانه ای کردن به دنیای نرم افزار های کامپیوتر راه یافته است. معماری نرم افزار نیز می تواند پیمانه ای باشد؛ یعنی نرم افزار به چند مؤلفهٔ جداگانه و متمایز تقسیم می شود که غالبا پیمانه خوانده می شود و از جامعیت آن ها خواسته های مسئله بر آورده می شود.

گفته شده است پیمانه ای، تنها صفتی از نرم افزار است که امکان مدیریت هوشمندانه یک برنامه را فراهم می کند. قابلیت خوانایی نرم افزار یکپارچه (یعنی برنامه بزرگی که تنها از یک پیمانه تشکیل شود) پایین است. تعداد مسیرهای کنترلی، تعداد متغیرها و پیچیدگی کلی باعث می شوند که درک برنامه تقریبا غیر ممکن شود.

برای نشان دادن این نکته، استدلال زیر را که مبتنی بر حل مسائل توسط انسان است، در نظر بگیرید:

فرض کنیم C(X) تابعی باشد که پیچیدگی مورد انتظار مسئله x و E(X) تابعی باشد که کار لازم برای حل مسئله x (بر حسب زمان) را تعیین می کند. برای دو مسئله p1 و p2، اگر:

$$C(P1) > E(P2) \qquad (۱-۱۳-الف)$$

خواهیم داشت:

$$E(P1) > E(P2) \qquad (۱-۱۳-ب)$$

این نتیجه یک چیز بدیهی است. حل مسئله ای که مشکلتر باشد زمان بیشتری می گیرد.

یک ویژگی جالب دیگر در حل مسئله توسط انسان کشف شده است:

یعنی

$$E(P1+P2) > E(p1)+E(P2) \qquad (۱۳-۳)$$

از معادله (۱۳-۳) چنین بر می آید که پیچیدگی مسئله ای مرکب از P1 و P2 بزرگتر از پیچیدگی تک تک این مسائل به صورت جداگانه است. با در نظر گرفتن معادله (۱۳-۲) و شرط معادله (۱۳-۱) خواهیم داشت :

$$E(P1+P2) > E(P1)+(P2) \qquad (۱۳-۳)$$

این موضوع منجر به نتیجه ای می شود که « تقسیم و حل» خوانده می شود- یعنی حل مسئله ای پیچیده با تقسیم آن به قطعات کوچکتر آسان تر می شود. نتایجی که در معادله (۱۳-۳) بیان شد در مورد پیمانه ای سازی و نرم افزار به خوبی مصداق پیدا می کند. این استدلال در واقع اثباتی برای پیمانه ای کردن است.

از معادله (۱۳-۳) می توان نتیجه گرفت که اگر نرم افزار را به طور نامتناهی به قطعات کوچکتر تقسیم کنیم، کار لازم برای آن به سمت صفر میل می کند! متأسفانه نیروهای دیگری نیز تأثیر می گذارند و باعث می شوند که این نتیجه (متأسفانه) اعتبار خود را از دست بدهد.

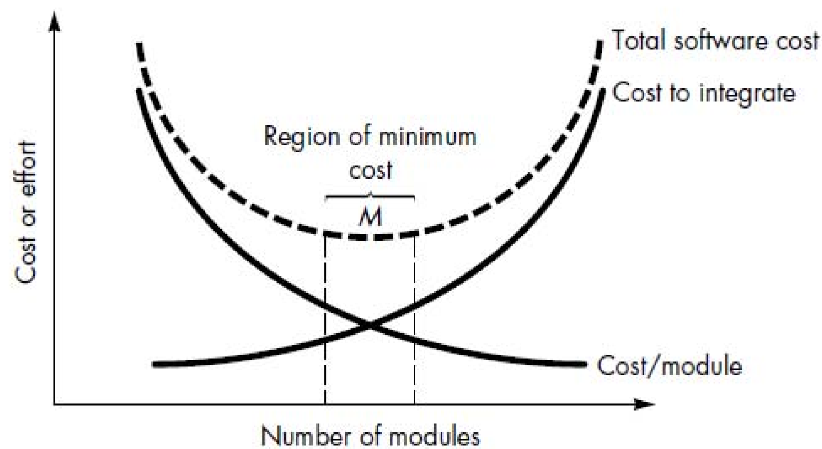
شکل ۱۳-۲ نشان می دهد که کار (هزینه) لازم برای یک پیمانۀ نرم افزاری با افزایش یافتن تعداد کل پیمانه ها حتما کاهش می یابد. اگر مجموعه خواسته ها یکسان نباشند، تعداد بیشتر پیمانه ها به معنای کاهش اندازۀ هر پیمانه است.

ولی به موازات رشد تعداد پیمانه ها، کار (هزینه) مربوط به جامعیت پیمانه نیز رشد می کند. این ویژگی ها منجر به منحنی هزینه یا کار کلی می شوند که در شکل نشان داده شده است. یک تعداد معین، M از پیمانه ها وجود دارد که به ازای آن، هزینه توسعه کمینه می شود. ولی لازم نیست خود را درگیر تعیین دقیق m کنیم.

منحنی های نشان داده شده در شکل ۱۳-۲ به هنگام پیمانه ای کردن بسیار مفید واقع می شوند. پیمانه ای کردن باید انجام شود ولی باید دقت شود که تعداد پیمانه ها در حول و حوش M باشد. ولی این حول وحوش M کجاست؟

نرم افزار تا چه حدی باید پیمانه ای باشد؟ پاسخ به این پرسش ها نیاز به درک مفاهیم دیگر طراحی دارد که بعداً در همین فصل شرح خواهیم داد.

هنگام در نظر گرفتن ویژگی پیمانه ای پرسش دیگری نیز مطرح می شود. یک پیمانه ای مناسب با اندازه ای مفروض را چگونه تعریف کنیم؟ پاسخ به روش (های) تعریف پیمانه های موجود در سیستم مربوط می شود. مایر پنج ملاک تعریف می کند که با استفاده از آن می توانیم توانایی یک روش طراحی را در تعریف سیستم پیمانه ای ارزیابی کنیم:



شکل ۲-۱۳ مدولاریته و هزینه نرم افزار

تجزیه پذیری پیمانه ای

اگر یک روش طراحی راهکاری سیستماتیک برای تجزیهٔ مسئله به مسائل کوچکتر فراهم آورد پیچیدگی کل مسئله کاهش یافته در نتیجه یک راهکار پیمانه ای کار آمد به دست می آید.

ترکیب پذیری پیمانه ای

اگر در یک روش طراحی بتوان مؤلفه های طراحی موجود (قابل استفاده ای مجدد) را در سیستم جدیدی مونتاژ کرد راهکاری پیمانه ای نتیجه می شود که در آن از دوباره کاری پرهیز شده است.

درک پذیری پیمانه ای

اگر پیمانه را بتوان به عنوان واحدی مستقل درک کرد (بدون رجوع به پیمانه های دیگر) آسانتر می توان آن را ساخت و تغییر داد.

تداوم پیمانه ای

اگر تغییرات کوچک در خواسته های سیستم منجر به تغییر دادن پیمانه های منفرد شوند (نه به تغییرات گسترده در کل سیستم) اثرات جانبی حاصل ازآن تغییر، کمترین ضربه را به پیکر سیستم وارد می آورد.

حفاظت پیمانه ای.

اگر در داخل پیمانه ای یک شرط نامطلوب رخ دهد و اثرات آن در آن پیمانه محدود شده باشد اثرات جانبی کاهش می یابد.

سرانجام لازم به ذکر است که یک سیستم را می توان پیمانه ای طراحی کرد حتی اگر لازم باشد که به صورت «یکپارچه» پیاده سازی شود. شرایطی وجود دارد (مثل نرم افزارهای بلادرنگ، نرم افزار های تعبیه شده) که در آنها سرعت نسبتا کمینه و سربار حافظه ناشی از زیربرنامه ها (زیرروالها و روالها) غیر قابل قبول است. در چنین مواردی نرم افزار را می توان و باید به صورت پیمانه ای طراحی کرد. ممکن است کد به صورت درونی توسعه یابد.

گر چه کد منبع برنامه ممکن است در نگاه نخست پیمانه ا ی به نظر نرسد این فلسفه حفظ شده است و برنامه از مزایای یک سیستم پیمانه ای برخوردار است.

فصل ۱۷: تکنیک های آزمون نرم افزار

آزمون نرم افزار چیست؟

هنگامی که کد منبع تولید شد، نرم افزار باید آزموده شود تا هر تعدادی از خطاها را که امکان داشته باشد، قبل از تحویل به مشتری کشف کرد. هدف، طراحی موارد آزمون است که احتمال یافتن خطا را بالا ببرند – ولی چگونه؟ اینجا است که تکنیکهای آزمون نرم افزار وارد صحنه می شوند. این تکنیکها راهنمایی سیستماتیک برای طراحی آزمونهایی به دست می دهند که: ۱- با منطق داخلی مولفه های نرم افزار تمرین می کنند و ۲. دامنه های داخلی و خارجی برنامه را تمرین می کنند تا خطاهای موجود در عملکرد، رفتار و کارآیی آن کشف شود.

چه کسی آن را انجام می دهد؟

طی مراحل اولیه آزمون، مهندس نرم افزار کلیه آزمونها را انجام می دهد. ولی به موازات پیشرفت فرآیند آزمون، ممکن است کارشناسان آزمون نیز در این امر شرکت کنند.

چرا اهمیت دارد؟

بازبینی ها و فعالیتهای SQA دیگر می توانند خطاها راکشف کنند و می کنند ولی کافی نیستند. هر بار که برنامه اجرا شود، مشتری آن را می آزماید ! بنابراین، باید برنامه را پیش از آنکه به دست مشتری برسد با هدف یافتن و حذف خطاها اجرا نمود. برای یافتن حداکثر تعداد ممکن خطاها، باید آزمونهایی را به طور سیستماتیک اجرا کرد و موارد آزمودنی را با استفاده از تکنیکهای منظم، طراحی نمود.

مراحل کار کدام است؟

نرم افزار از دو دیدگاه متفاوت آزمایش می شود:

۱-با استفاده از تکنیکهای طراحی موارد آزمون با منطق داخلی برنامه (جعبه سفید) تمرین می شود.

۲- با خواسته های نرم افزار با استفاده از تکنیکهای طراحی مورد آزمون « جعبه سیاه » تمرین می شود. در هر دو حال، هدف، یافتن حداکثر تعداد خطاها با حداقل مقدار کار و زمان است.

محصول کاری چیست؟

مجموعه ای از موارد آزمون که برای تمرین با منطق داخلی و خواسته های خارجی، طراحی و مستندسازی شده اند؛ نتایج مورد انتظار تعیین و نتایج واقعی ثبت می شوند.

چگونه مطمئن شوم که درست از عهده کارها برآمده ام؟

هنگامی که آزمون را شروع می کنید، دیدگاه خود را تغییر دهید. موارد آزمون را به شیوه ای منظم طراحی کنید و آنها را بازبینی کنید تا مطمئن شوید که کامل هستند.

۱-۱۷- مبانی آزمون نرم افزار

آزمون برای مهندس نرم افزاری امری غیر عادی تلقی می شود. طی مراحل اولیه تعریف و توسعه، مهندس می کوشد تا نرم افزار را از یک مفهوم انتزاعی، یک محصول عینی بسازد. اکنون نوبت به آزمون می رسد. مهندس، مواد آزمون را به منظور تخریب نرم افزار ساخته شده طراحی می کند. در حقیقت، آزمون، تنها مرحله ای از فرآیند نرم افزار است که بیشتر ویرانگر (حداقل از نظر روان شناختی) به نظر می رسد تا سازنده.

مهندسان نرم افزار، فطرتاً افرادی سازنده هستند. آزمون، مستلزم آن است که سازنده دید پیش داوری خود، مبنی بر درستی نرم افزار را دور بریزد و بر تناقص جالبی که از کشف خطاها عارض می شود، غلبه کند. بیزر [BE190] این وضعیت را به خوبی شرح مید هد:

اسطوره ای وجود دارد مبنی بر اینکه اگر واقعاً در برنامه نویسی استاد باشیم، اشکالی وجود نخواهد داشت. اگر فقط می توانستیم واقعاً حواس خود را جمع کنیم، اگر فقط همه از برنامه نویسی ساخته یافته، طراحی بالا به پایین و جداول تصمیم گیری استفاده می کردند، اگر برنامه به SQUISH نوشته می شد، و اگر بخت با ما یار بود، هیچ اشکالی پیش نمی آمد و این اسطوره همچنان ادامه دارد. این اسطوره می گوید اشکالات از آن رو وجود دارند که ما کارها را خوب انجام نمی دهیم و اگر خوب انجام نمی دهیم باید درباره آن احساس گناه کنیم. بنابراین، آزمودن و طراحی موارد آزمون، پذیرش شکست است که رفته رفته به پذیرش گناه می انجامد. کسالت امر آزمون، تنبیهی برای خطاهای ما است. تنبیه برای چه؟

برای انسان؟ گناه برای چه؟ برای شکست در دستیابی به کمالات فرا انسانی؟ برای اینکه نتوانیم بین آنچه که یک برنامه نویس دیگر می گوید و می اندیشد تمایز قائل شویم؟ برای حل نکردن مشکلات ارتباطی انسانی که قرنها است لاینحل مانده است؟

آیا آزمون باید به پذیرش گناه بینجامد؟ آیا آزمون واقعاً ویرانگر است؟ پاسخ این پرسشها منفی است. ولی اهداف آزمون تا حدی متفاوت با انتظار ماست.

۱-۱-۷ اهداف آزمون

گلن مایرز در یک کتاب عالی درباره نرم افزار [MYE79] چند قاعده را بیان می کند که به خوبی به عنوان اهداف آزمون عمل می کند.

۱.آزمون، فرآیند اجرای برنامه به قصد یافتن خطاست.

۲.مورد آزمون خوب، موردی است که احتمال یافتن خطاهای کشف شده در آن، بالا باشد.

۳. آزمون موفق، آزمونی است که خطاهای کشف نشده را کشف می کند.

اهداف بالا نشانگر یک تغییر دیدگاه زیبا هستند و برخلاف این دیدگاه عامیانه که آزمون موفق، آزمونی است که در آن خطایی یافته نشود.

اگر آزمون با موفقیت اجرا شود (مطابق اهداف ذکر شده در بالا)، خطاهای نرم افزار را برملا خواهد نمود. به عنوان مزیت دوم، آزمون نشان می دهد که عملکردهای نرم افزار ظاهراً مطابق مشخصه کار می کنند و خواسته های رفتاری و کارآیی ظاهراً برآورد شده اند. به علاوه، داده های جمع آوری شده به موازات انجام آزمون، شاخص خوبی از قابلیت اطمینان نرم افزار و شاخصی از کلیت کیفیت نرم افزار به دست می دهند. ولی آزمون نمی تواند نبود خطاها و نقایص را ثابت کند. بلکه فقط می تواند نشان دهد که خطاها و نقایص وجود دارند.

۲-۱-۱۷- اصول آزمون

مهندس نرم افزار پیش از اعمال روشها در خصوص موارد آزمون موثر، باید اصول پایه ای را که آزمون نرم افزار را هدایت می کنند، درک کند. دیویس [DAV 95] مجموعه ای از اصول پیشنهاد می کند که در این کتاب از آنها استفاده خواهیم کرد:

- **همه آزمونها باید تا حد خواسته های مشتری قابل ردیابی باشند**. چنانکه دیدیم، هدف آزمون نرم افزار، کشف خطاها است. یعنی اکثر نقایص شدید (از دیدگاه مشتری) آنهایی هستند که باعث می شوند برنامه نتواند خواسته های خود را برآورده کند.

- **آزمون باید مدتها قبل از شروع آزمون، طرح ریزی شود**. طرح ریزی آزمون می تواند به محض کامل شدن مدل خواسته ها آغاز شود. تعریف مشروح موارد آزمون می تواند به محض منسجم شدن مدل طراحی آغاز شود. بنابراین، همه آزمونها را می توان پیش از تولید هرگونه کد، برنامه ریزی و طراحی کرد.

- **اصل پارتو در آزمون نرم افزار صدق می کند**. به عبارت ساده، اصل پارتو بیان می کند که ۸۰ درصد همه خطاهای کشف شده طی آزمون، احتمالاً در ۲۰ درصد همه مولفه های برنامه قابل کشف هستند. مسئله، جداسازی مولفه های مظنون و آزمودن کامل آنهاست.

- **آزمون باید در ابعاد کوچک آغاز شود و به ابعاد بزرگتر گسترش یابد**. اولین آزمونها بر روی هر یک از مولفه های انجام می شوند. با پیشرفت آزمون، خطاهای مجموعه ای از مولفه های مجتمع و سپس کل سیستم یافته می شود.

- **برای آنکه آزمون بیشترین بازدهی را داشته باشد، باید توسط یک شخص ثالث بی طرف انجام شود.** منظور از بیشترین بازدهی آن است که خطاها را با احتمال بیشتری بیابد، به دلایلی که قبلاً در همین فصل ذکر شد، مهندس نرم افزاری که سیستم را ایجاد کرده است، بهترین کسی نیست که باید همه آزمونها را انجام دهد.

۳-۱-۱۷ – آزمون پذیری

در شرایط ایده آل، مهندس نرم افزار یک برنامه کامپیوتری، یک سیستم یا یک محصول را طوری طراحی می کند که آزمون پذیری داشته باشد، به این ترتیب افرادی که آزمون را انجام می دهند، می توانند موارد آزمون کارآمد را آسانتر طراحی کنند. آزمون پذیری چیست؟ جیمزبک، آزمون پذیری را به شیوه زیر تعریف می کند:

آزمون پذیری نرم افزار، میزانی از سهولت آزمودن یک برنامه کامپیوتری است. چون آزمون دشوار است بهتر است بدانیم چه چیزی آن را به جریان می اندازد. گاهی برنامه نویسان مشتاق به انجام کارهایی هستند که به فرآیند آزمون کمک می کند و لیست کنترلی از نکات طراحی، ویژگیها و غیره می تواند در بحث با آنها مفید واقع شود.

معیارهایی وجود دارد که جنبه های مختلف آزمون پذیری را اندازه گیری می کند، گاهی آزمون پذیری به این معنا است که چگونه مجموعه خاصی از آزمونها، محصول را تحت پوشش قرار می دهد. در ارتش به معنای این است که یک ابزار چگونه در میدان جنگ قابل کنترل و ترمیم است. این دو معنا " قابلیت آزمون نرم افزار " را توصیف نمی کنند. لیست کنترلی که در ادامه می آید، ویژگیهایی را مطرح می کند که نرم افزار آزمون پذیر را ایجاد می نماید:

قابلیت کار.« هر چه بهتر کار کند، آزمون آن کارآمدتر است.»

- سیستم چند اشکال معدود دارد (این اشکالات باعث افزایش تحلیل و گزارش دهی به فرآیند آزمون می شود.)
- هیچ اشکالی مانع اجرای آزمونها نمی شود.
- محصول در مراحل عملیاتی تکامل می یابد (امکان توسعه و آزمون همزمانی وجود دارد.)

قابلیت مشاهده. « آنچه می بینید، همان است که آزمایش می کنید. »

- برای هر ورودی یک خروجی متمایز تولید می شود.
- متغیرها و حالتهای سیستم در اثنای اجرا قابل مشاهده و استفسارند.
- متغیرها و حالتهای گذشته سیستم قابل مشاهده و استفسارند (مثل ثبت وقایع تراکنشی)
- همه وقایعی که خروجی را تحت تاثیر قرار می دهند، قابل مشاهده اند.
- خروجی نادرست به آسانی قابل شناسایی است.
- خطاهای داخلی به طور خودکار از طریق راهکارهای خودآزمایی آشکار می شوند.
- خطاهای داخلی به طور خودکار گزارش می شوند.

- کد منبع قابل دستیابی است

کنترل پذیری. «هر چه بهتر بتوان نرم افزار را کنترل کرد، آزمون را بیشتر می توان خودکار و بهینه کرد.»

- همه خروجیهای ممکن را می توان از طریق ترکیبی از ورودیها تولید نمود.
- همه کدها از طریق ترکیبی از ورودیها قابل اجرا هستند.
- متغیرها و حالتهای سخت افزاری و نرم افزاری مستقیماً توسط مهندس آزمودن قابل کنترل هستند.
- فرمتهای خروجی و ورودی، سازگار و ساخت یافته هستند.
- آزمونها را به راحتی می توان مشخص، خودکار و بازسازی کرد.

تجزیه پذیری: « با کنترل دامنه کاربرد آزمون، می توان مسائل را سریعتر جداسازی کرد و آزمونهای مجدد رابا هوشمندی بیشتر انجام داد. »

- سیستم نرم افزاری از پیمانه های مستقل ساخته می شود.
- پیمانه های نرم افزاری را می توان مستقل از هم آزمود.

سادگی.« هرچه مورد آزمون کوچکتر باشد، سریعتر می توان آن را آزمود. »

- سادگی عملیاتی (مثلاً مجموعه ویژگیها، حداقل مجموعه لازم برای برآوردن خواسته ها است.)
- سادگی ساختاری (مثلاً معماری به صورت پیمانه ای درآمد تا انتشار خطاها را به حداقل برساند).
- سادگی کد (مثلاً یک استاندارد کد نویسی رعایت می شود که وارسی و نگهداری آن آسان باشد.)

پایداری. « هر چه تعداد تغییرات کمتر باشد، آزمون کمتر با مانع مواجه می شود. »

- تغییرات نرم افزار چندان زیاد نیست.
- تغییرات نرم افزار کنترل شده هستند.
- تغییرات نرم افزار آزمونهای موجود را بی اعتبار نمی کنند.

- نرم افزار به خوبی از پس شکستها بر می آید.

درک پذیری « هر چه اطلاعات بیشتری داشته باشیم، آزمون هوشمندانه تر انجام می شود»

- طراحی به خوبی درک شده است
- وابستگی میان مولفه های داخلی، خارجی و مشترک به خوبی درک شده است.
- تغییرات در طراحی مورد تبادل قرار گرفته اند.
- مستندات فنی بلافاصله قابل دستیابی اند.
- مستندات فنی به خوبی سازمان یافته اند.
- مستندات فنی مشخص و مفصل هستند.
- مستندات فنی صحیح هستند.

مهندسی نرم افزار می تواند با استفاده از صـــفاتی که بَک توصیه می کند، پیکربندی نرم افزار (یعنی برنامه ها، داده ها و مستندات) را طوری توسعه دهد که مستعد آزمون باشد.

درباره خود آزمونها چه می توان گفت؟ کینر، فالک و نگورین [KAN93] صفات زیر را برای یک آزمون « خوب » بر می شمرند.

۱. آزمون خوب با احتمال زیادی خطاها را می یابد، برای حصول این منظور، آزمونگر باید نرم افزار را بشناسد و کوشش کند تا یک تصویر ذهنی از چگونگی شکست احتمالی نرم افزار بسازد.

۲. آزمون خوب دارای زواید نیست. زمان و منابع آزمون، محدود است. در اجرای آزمونی که هدف آن با هدف یک آزمون دیگر یکی است هیچ نکته ای وجود ندارد. هر یک از آزمونها باید دارای هدفی متفاوت باشد (حتی اگر این تفاوت ظریف باشد.)برای مثال، پیمانه ای از نرم افزار SafeHome برای شناسایی کلمه عبور کاربر جهت فعال و غیر فعال کردن سیستم طراحی می شود. آزمونگر برای کشف خطا در ورود کلمه عبور، آزمونهایی را طراحی می کند که دنباله ای از کلمه های عبور را وارد می کنند. کلمه های عبور معتبر و

نامعتبر (رشته های چهاررقمی) به عنوان آزمونهایی جداگانه وارد می شوند. ولی هر کلمه عبور معتبر / نامعتبر باید حالت متفاوتی از شکست را منعکس کند. برای مثال، کلمه عبور نامعتبر 1234 نباید توسط سیستمی پذیرفته شود که طبق برنامه ریزی، کلمه عبور معتبر 8080 را می پذیرد. اگر پذیرفته شود، خطا وجود دارد. یک ورودی آزمایشی دیگر مثلاً ۱۲۳۵ همان هدف ۱۲۳٤ را دنبال می کند و لذا زاید است، ولی ورودیهای نامعتبر همچون ۸۰۸۱ یا ۸۱۸۰ دارای تفاوتی ظریف بوده کوشش می کنند تا نشان دهند که برای کلمات عبور نزدیک به کلمه عبور معتبر، ولی نه همسان با آنها، خطایی وجود دارد.

۳. یک آزمون خوب باید « بهترین » باشد [KAN93]. در گروهی از آزمونها که هدف و قصدی مشابه دارند، محدودیتهای منابع و زمانی ممکن است فقط با اجرای فقط زیر مجموعه ای از این آزمونها تعدیل شوند. در چنین مواردی، آزمونی به کار گرفته می شود که با احتمال بیشتری خطا را می یابد.

۴. آزمون خوب نباید بیش از حد ساده و نه بیش از حد پیچیده باشد. گرچه ممکن است تعدادی آزمون را در یک آزمون خلاصه کرد، اثرات جانبی این روش ممکن است خطاها را نادیده بگیرد. به طور کلی، هر آزمون باید جداگانه اجرا شود.

۱۷-۲ – طراحی موارد آزمون

طراحی آزمونها برای نرم افزار و هر محصول مهندسی دیگر، می تواند به اندازه طراحی خود محصول اولیه دشوار باشد. با این حال به دلایلی که پیش از این بحث شد، مهندسان نرم افزار غالباً با آزمون، به عنوان موارد آزمون در حال توسعه ای رفتار می کنند که ممکن است درست به نظر آیند، ولی ازکامل بودن آنها چندان اطمینانی نباشد. با به خاطر داشتن اهداف آزمون، باید آزمونهایی را طراحی کنیم که احتمال یافتن خطاها در حداقل زمان، بیشتر باشد.

هر محصول مهندسی (و اکثر چیزهای دیگر) را می توان به یکی از دو روش آزمایش کرد:

۱- با دانستن عملکرد خاصی که نرم افزار برای انجام آن طراحی شده است، می توان آزمونهایی طراحی کرد که نشان می دهند هر عملکرد به طور کامل درست است، ودر عین حال، در هر عملکرد به دنبال یافتن خطاها هستند.

۲- با دانستن طرز کار داخلی محصول، می توان آزمونهایی ترتیب داد که اطمینان دهند همه چیز جفت و جور است. یعنی، عملیات داخلی طبق مشخصه اجرا می شوند و با همه مولفه های داخلی به طور مناسب تمرین شده است. روش اول را آزمون جعبه سیاه و دومی را آزمون جعبه سفید می نامند.

در آزمون نرم افزارهای کامپیوتری منظور از آزمون جعبه سیاه آزمونهایی است که در واسط نرم افزار اجرا می شوند. گرچه تستهای جعبه سیاه برای کشف خطاها طراحی شدند، برای این اهداف نیز به کار می روند: آیا عملکردهای نرم افزار، انجام پذیر هستند، ورودی به طور مناسب پذیرفته شده است، و خروجی به درستی تولید شده است؛ آیا جامعیت اطلاعات خارجی (مثل بانک اطلاعاتی) حفظ شده است یا خیر. آزمون جعبه سیاه جنبه های بنیادی سیستم را بدون در نظر گرفتن ساختار منطقی داخلی نرم افزار، مورد بررسی قرار می دهد.

آزمون جعبه سفید نرم افزار بررسی دقیق جزئیات رویه ای را در بر می گیرد، مسیرهای منطقی در سرتاسر نرم افزار آزمون می شود؛ برای این منظور، موارد آزمونی طراحی می شود که روی مجموعه مشخصی از شرایط و یا حلقه ها کار می کنند. وضعیت برنامه ممکن است در نقاط گوناگون مورد بررسی قرار گیرد تا تعیین شود که آیا وضعیت مورد انتظار یا ارزیابی شده با وضعیت واقعی شباهت دارد یاخیر.

در نگاه نخست، ممکن است به نظر برسد که آزمون کامل جعبه سفید منجر به برنامه های ۱۰۰٪ درست شود. کافی است همه مسیرهای منطقی را تعیین کنیم، موارد آزمون را برای آنها پایه ریزی و نتایج را ارزیابی کنیم، یعنی موارد آزمون جامعی برای امتحان کردن منطق برنامه تولید کنیم. متاسفانه، آزمون جامع مشکلات منطقی خاصی را ایجاد می کند. حتی برای برنامه های کوچک، تعداد مسیرهای منطقی می تواند بسیار بزرگ باشد. برای مثال، یک برنامه ۱۰۰ خطی به زبان C را در نظر بگیرید. پس از اعلان داده های اصلی، برنامه حاوی دو حلقه تودرتو است که هر کدام بسته به ورودی اولیه، ۱ تا ۲۰ بار اجرا می شود. در داخل حلقه اول، چهار ساختار if- then –else مورد نیاز است. در چنین برنامه ای حدود $۱۰^{۱۴}$ مسیر ممکن وجود دارد !

برای آنکه تصویری از این عدد به دست آورید، فرض کنیم که یک پردازنده سحرآمیز (سحرآمیز از آنرو که چنین پردازنده ای وجود ندارد) برای آزمون جامع ساخته شده باشد این پردازنده می تواند توسعه یک مورد آزمون، اجرای آن و ارزیابی نتایج حاصل از اجرای آن را در مدت یک میلی ثانیه کامل کند. این پردازنده اگر روزی ۲۴ساعت

به هرحال نباید این سوءِ تعبیر پیش آید که آزمون جعبه سفید، غیر عملی است. تعداد محدودی از مسیرهای منطقی را می توان انتخاب و با آنها تمرین کرد. ساختمان داده های مهم را می توان بررسی کرد و از اعتبار آنها اطمینان یافت. صفات هر دو روش جعبه سیاه و جعبه سفید را می توان در هم آمیخت و روشی فراهم آورد که واسط نرم افزار را اعتبار سنجی کند و به طور گزینشی اطمینان دهد که عملیات داخلی نرم افزار صحیح هستند.

۳-۱۷-آزمون جعبه سفید

آزمون جعبه سفید که گاه جعبه شیشه ای نیز خوانده می شود، یک روش طراحی برای موارد آزمونی است که برای به دست آوردن موارد آزمون از ساختار کنترلی طراحی رویه ای استفاده می کند. مهندس نرم افزار با استفاده از متدهای آزمون جعبه سفید می تواند موارد آزمونی به دست آورد که ۱.تضمین می کنند که همه مسیرهای مستقل در یک پیمانه حداقل یک بار امتحان شده اند؛ ۲.همه تصمیم گیریهای منطقی را در دو بخش درست و غلط امتحان کنند؛ ۳.همه حلقه ها را در مرزها و در داخل مرزهای عملیاتی آنها اجرا کنند؛ و ٤.ساختمان داده های داخلی را امتحان کنند تا اعتبار آنها ثابت شود.

در این خصوص یک پرسش منطقی ممکن است مطرح شود: «چرا باید وقت و انرژی خود را صرف جزئیات منطقی (آزمودن آنها) کنیم، در حالی که می توان این انرژی را صرف حصول اطمینان از برآورده شدن خواسته ها کرد؟ » به بیان دیگر، چرا همه انرژی خود را صرف آزمونهای جعبه سیاه نکنیم؟ پاسخ در ماهیت نقایص نرم افزاری نهفته است [JON81]:

خطاهای منطقی و فرضیات نادرست، با احتمال اینکه یک مسیر از برنامه ای اجرا شود نسبت عکس دارند. هنگامی که یک عملکرد، شرایط یا کنترل خارج از جریان اصلی را طراحی و پیاده سازی می کنیم، ممکن است دچار خطا شویم. پردازش تکراری (هرروزه) معمولا به خوبی درک می شود، در حالی که در پردازش «موارد خاص » معمولا مشکل پیش می آید.

غالبا بر این باوریم که یک مسیر منطقی خاص، همواره به طور منظم اجرا می شود. جریان منطقی یک برنامه گاهی هوشمندانه نیست یعنی فرضیات ناخودآگاه ما درباره جریان کنترل و داده ها ممکن است باعث شود که دچار خطاهای طراحی شویم که فقط هنگام شروع آزمون مسیر کشف می شوند.

خطاهای تایپی ماهیتی تصادفی دارند. هنگامی که برنامه ای به کد منبع زبان برنامه نویسی ترجمه می شود این احتمال وجود دارد که برخی خطاهای تایپی رخ دهد. بسیاری از آنها توسط راهکارهای چک کننده تایپ و فرمت کشف می شوند ولی بقیه ممکن است تا زمان شروع آزمون کشف نشوند. این احتمال وجود دارد که یک خطای تایپی در یک مسیر منطقی مبهم وجود داشته باشد.

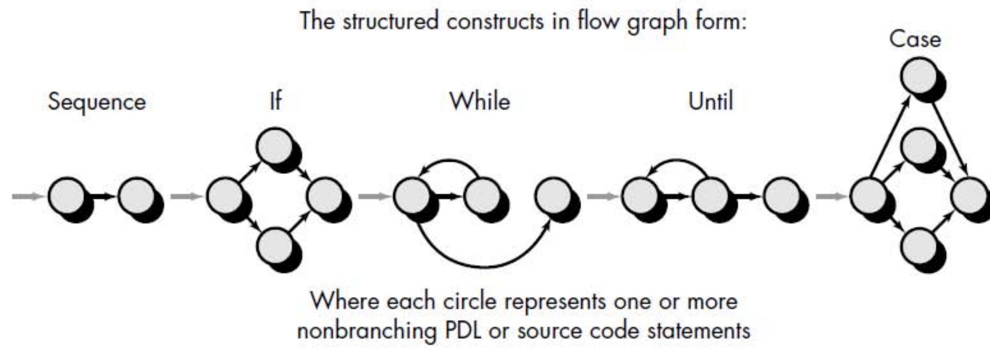
٤-١٧- آزمون مسیرهای پایه

آزمون مسیر، مبنای یک تکنیک آزمون جعبه سفید است که نخستین بار توسط نام مک کیب [MCC76] پیشنهاد شد. روش مسیرهای پایه، طراح موارد آزمون را قادر می سازد تا میزانی منطقی از پیچیدگی رویه ای به دست آورد و از این میزان به عنوان راهنمایی جهت تعریف یک مجموعه پایه از مسیرهای اجرا استفاده کند. موارد آزمون به دست آمده برای امتحان کردن این مجموعه پایه، هر دستور از برنامه را حداقل یک بار در اثنای آزمون اجرا خواهند کرد.

١-٤-١٧نشانه گذاری گراف جریان

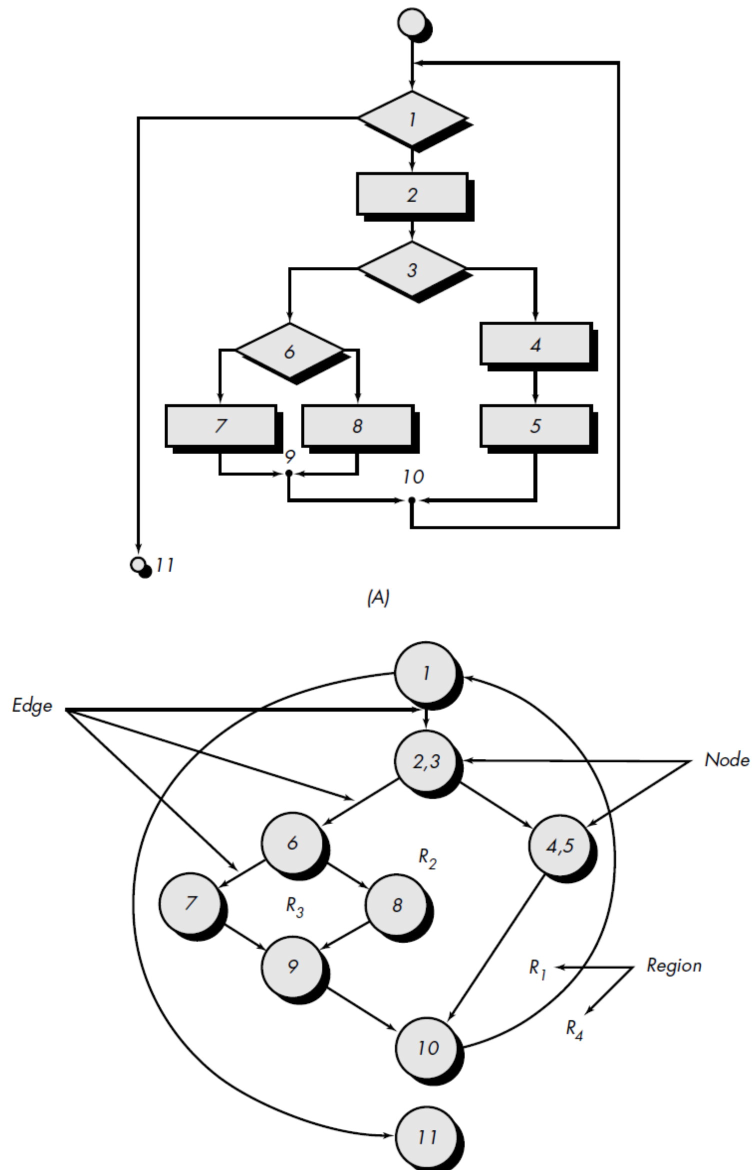
پیش از آنکه بتوان روش مسیرهای پایه را معرفی نمود، یک نشانه گذاری ساده، موسوم به گراف جریان یا (گراف برنامه) را باید برای نمایش دادن جریان معرفی کرد. گراف جریان، جریان کنترل منطقی را با استفاده از نشانه گذاری درشکل ١-١٧تصویر می کند. متناظر با هر ساختار ساخت یافته یک نماد گراف جریان وجود دارد.

FIGURE 17.1
Flow graph notation

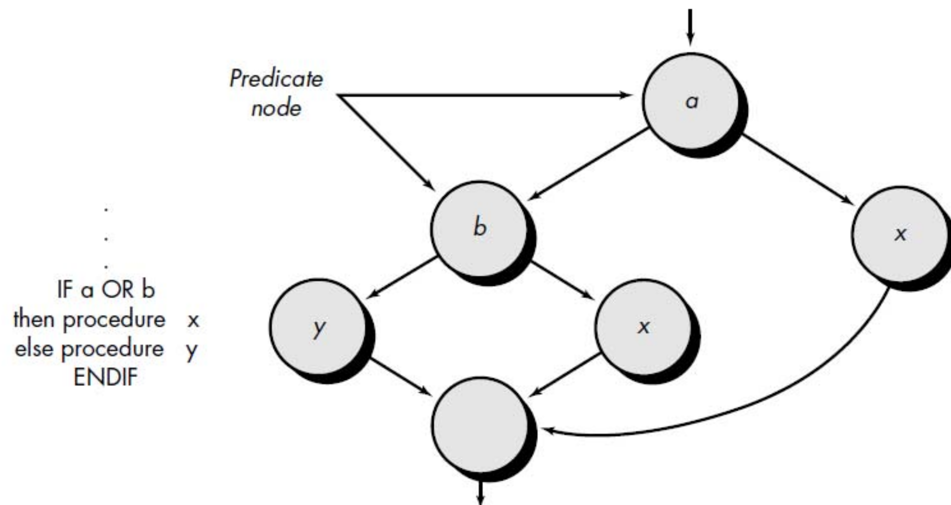


شکل ۱-۱۷ نشانه گذاری گراف گردشی

FIGURE 17.2
Flowchart, (A) and flow graph (B)



(B) شکل ۲-۱۷ الف نمودار گرشی ب گراف جریان



شکل ۳-۱۷ منطق مرکب

برای نشان دادن کاربرد گراف جریان، طراحی رویه ای شکل۲-۱۷-الف را در نظر می گیریم. در اینجا، از یک نمودار گردشی برای تصویر کردن ساختار کنترلی برنامه استفاده می شود. شکل ۲-۱۷-ب نمودار گردشی را به صورت گراف جریان آن در آورده است (با این فرض که هیچ شرط مرکبی در لوزیهای تصمیم گیری نمودار گردشی وجود نداشته باشد). با توجه به شکل ۲-۱۷-ب هردایره که گره گراف جریان خوانده می شود یک یا چند دستور رویه ای را نشان می دهد. ترتیبی از مستطیلهای پردازشی و یک لوزی تصمیم گیری را می توان در یک گروه منفرد خلاصه نمود. پیکان های روی گراف جریان که یال یا پیوند خوانده می شوند، نشانگر جریان کنترل بوده مشابه پیکان های نمودار گردشی هستند هر رویه باید در یک گره پایان یابد، حتی اگر گره هیچ دستور رویه ای را نشان ندهد (مثلا نماد مربوط به ساختمان if-then-else ببینید)مساحتهای محصور شده توسط یالها و گره ها را ناحیه می نامند هنگام شمارش نواحی مساحت خارج از گراف را نیز به عنوان یک ناحیه در نظر گرفته آن را لحاظ می کنیم.

هنگام مواجهه با شرطهای مرکب در یک طراحی رویه ای، تولید گراف جریان قدری پیچیده تر می شود. شرط مرکب زمانی رخ می دهد که یک یا چند عملگر بولی (NAND,NOR,AND,OR منطقی) در یک دستور شرطی وجود داشته باشند. در شکل ۳-۱۷ دستور PDL به گراف جریان ترجمه می شود. توجه داشته باشید که برای هریک از شرایط a و b در دستورIF a OR b یک گروه جداگانه ایجاد می شود. هرگره که حاوی یک شرط باشد، گره گزاره ای خوانده می شود و با دو یا چند پیکان که از آن بیرون می آیند، مشخص می شود.

۲-۱۴-۱۷پیچیدگی سیکلوماتیک

پیچیدگی سیکلوماتیک یک معیار نرم افزاری است که میزانی کمی از پیچیدگی منطقی یک نرم افزار ارائه می دهد. در روشهای آزمون پایه، مقدار محاسبه شده برای پیچیدگی سیکلومات تعداد مسیرهای مستقل در مجموعه پایه یک برنامه را تعیین می کند و یک حد فوقانی برای تعداد مسیرهایی فراهم می آورد که باید اجرا شوند تا اطمینان حاصل شود که همه دستورها حداقل یک بار اجرا شده اند.

مسیر مستقل، هر مسیری از برنامه است که حداقل یک مجموعه جدید از دستورهای پردازش یا یک دستور شرطی را معرفی کند. اگر مسیر مستقل بر حسب گراف جریان بیان شود، حداقل باید در راستای یک یال حرکت کند که پیش از تعریف مسیر از آن عبور نشده باشد. برای مثال، مجموعه ای از مسیرهای مستقل در شکل ۲-۱۷-ب در گراف جریان نشان داده شده اند:

مسیر ۱ :۱-۱۱

مسیر ۲ :۱-۲-۳-٤-۵-۱۰-۱-۱۱

مسیر ۳ :۱-۲-۳-۶-۸-۹-۱۰-۱-۱۱

مسیر٤ :۱-۲-۳-۶-۷-۹-۱۰-۱-۱۱

توجه دارید که هر مسیر جدید یک یال جدید معرفی می کند. مسیر زیر:

۱-۲-۳-٤-۵-۱۰-۱-۲-۳-۶-۸-۹-۱۰-۱-۱۱

به عنوان مسیری مستقل در نظر گرفته نمی شود زیرا صرفا تلفیقی از مسیرهای مشخص شده از قبل است و از هیچ یال جدیدی عبور نمی کند.

مسیرهای ۱،۲،۳،٤ که در بالا تعریف شده اند یک مجموعه پایه را برای گراف جریان شکل ۲-۱۷-ب تشکیل می دهند یعنی اگر آزمونها را بتوان طوری طراحی کرد که اجرای این مسیرها (یک مجموعه پایه) را حتمی کنند هر دستور از برنامه حداقل یک بار اجرا خواهد شد و هر دستور شرطی در هر دو حالت درست و نادرست خود اجرا می شود. لازم به ذکر است که مجموعه پایه منحصر بفرد نیست. در حقیقت چند مجموعه پایه متفاوت را می توان برای یک طراحی رویه ای مفروض به دست آورد.

چگونه باید بدانیم که به دنبال چند مسیر باید بگردیم؟ پاسخ را در محاسبه پیچیدگی سیکلوماتیک می توان جستجو کرد.

پیچیدگی سیکلوماتیک ریشه در نظریه گرافها دارد و یک معیار نرم افزاری سودمند را فراهم می آورد.

پیچیدگی به یکی از سه شیوه زیر محاسبه می شود:

۱.تعداد نواحی گراف جریان متناظر با پیچیدگی سیکلوماتیک؛

۲.پیچیدگی سیکلوماتیک برای یک گراف جریان V(G) به صورت زیر تعریف می شود

$$V(G)=E-N+2،$$

که درآن E تعداد یالهای گراف جریان و N تعداد گره های آن است.

۳.پیچیدگی سیکلوماتیک V(G) برای گراف جریان G به صورت زیر تعریف می شود:

$$V(G)=P+1$$

که در آن P تعداد گره های گزاره ای موجود در گراف جریان G است.

اگر یک بار دیگر به گراف جریان شکل ۲-۱۷-ب رجوع کنید، می بینید که پیچیدگی سیکلوماتیک را می توانید با به کارگیری هر یک از الگوریتمهای بالا محاسبه کنید:

۱- گراف گردشی چهار ناحیه دارد.

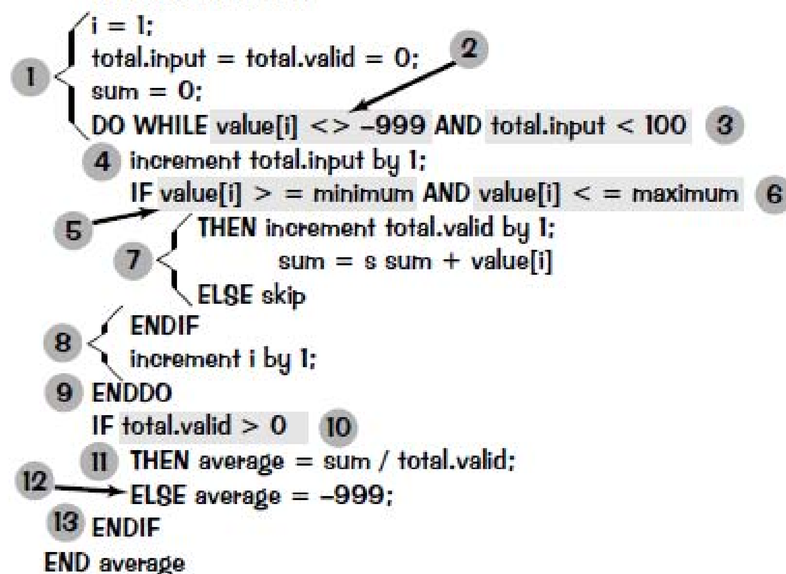
۲- V(G)=یال۱۱ -گره ۹+۲=٤

۳- V(G)= گره گزاره ای ۳+۱=٤

بدین ترتیب پیچیدگی سیکلوماتیک گراف جریان در شکل ۲-۱۷-ب برابر ٤ است.

مهمتر اینکه مقدار V(G) یک حد فوقانی برای تعداد مسیرهای تشکیل دهنده مجموعه پایه ارائه می کند و در نتیجه یک حد فوقانی برای تعداد آزمون نهایی ارائه می کند که باید طراحی و اجرا شوند تا تضمین شود که کلیه دستورات برنامه تحت پوشش قرار گرفتند.

FIGURE 17.4
PDL for test case design with nodes identified



```
PROCEDURE average;

    *  This procedure computes the average of 100 or fewer
       numbers that lie between bounding values; it also computes the
       sum and the total number valid.

    INTERFACE RETURNS average, total.input, total.valid;
    INTERFACE ACCEPTS value, minimum, maximum;

    TYPE value[1:100] IS SCALAR ARRAY;
    TYPE average, total.input, total.valid;
       minimum, maximum, sum IS SCALAR;
    TYPE i IS INTEGER;
 1  i = 1;
 1  total.input = total.valid = 0;            2
 1  sum = 0;
 1  DO WHILE value[i] <> -999 AND total.input < 100   3
 4     increment total.input by 1;
       IF value[i] > = minimum AND value[i] < = maximum   6
 5  7      THEN increment total.valid by 1;
 7              sum = s sum + value[i]
 7         ELSE skip
 8     ENDIF
 8     increment i by 1;
 9  ENDDO
    IF total.valid > 0   10
 11    THEN average = sum / total.valid;
 12    ELSE average = -999;
 13 ENDIF
END average
```

شکل ٤-١٧ PDL برای طراحی موارد آزمون که در آن گره ها تعیین شده است

٣-٤-١٧ به دست آوردن موارد آزمون

روش آزمون مسیرهای پایه را می توان در یک طراحی رویه ای یا کد منبع به کار برد. در این بخش آزمون مسیرهای پایه را به صورت چند مرحله ارائه خواهیم داد. از رویه average در شکل ٤-١٧ تصویر شده است به عنوان مثالی برای نشان دادن هر یک از مراحل این روش استفاده خواهیم کرد. توجه داشته باشید که گرچه average یک الگوریتم بسیار ساده است، حاوی حلقه ها و شرطهای مرکب است. برای به دست آوردن مجموعه پایه باید مراحل زیر را اجرا نمود:

۱.استفاده از طراحی یا کد به عنوان یک بستر و رسم گراف جریان مربوطه. گراف جریان با استفاده از نمادهای قواعد ذکر شده در بخش ۱-٤-۱۷ ایجاد می شود. با توجه به PDL مربو ط به رویه average در شکل ٤-۱۷ گراف جریان با شماره گذاری آن دسته از دستورهای PDL ایجاد می شود که در گره های گراف جریان مربوط تصویر شوند.

۲. پیچیدگی سیکلوماتیک گراف جریان حاصل را تعیین کنید. پیچیدگی سیکلوماتیک V(G) با اعمال الگوریتم های تشریح شده در بخش ۲-۱۵-۷ تعیین می شود. باید توجه داشته باشید که V(G) را می توان بدون توسعه یک گراف جریان، با شمارش کلیه دستورهای شرطی در PDL (برای رویه average شرایط ترکیبی برابر با ۲ است) و افزودن یک واحد، محاسبه کرد.

V(G)= ناحیه ۶

V(G)= یال ۱۷-گره ۳+۲=۶

V(G)= گره گزاره ای ۵+۱=۶

۳.تعیین مجموعه پایه برای مسیرهای مستقل خطی. مقدار V(G) با استفاده از تعداد مسیرهای مستقل خطی موجود در ساختار کنترلی برنامه مشخص می شود در مورد رویه average انتظار داریم شش مسیر مشخص شود:

مسیر۱: ۱-۲-۱۰-۱۱-۱۳

مسیر۲: ۱-۲-۱۰-۱۲-۱۳

مسیر۳: ۱-۲-۳-۱۰-۱۱-۱۳

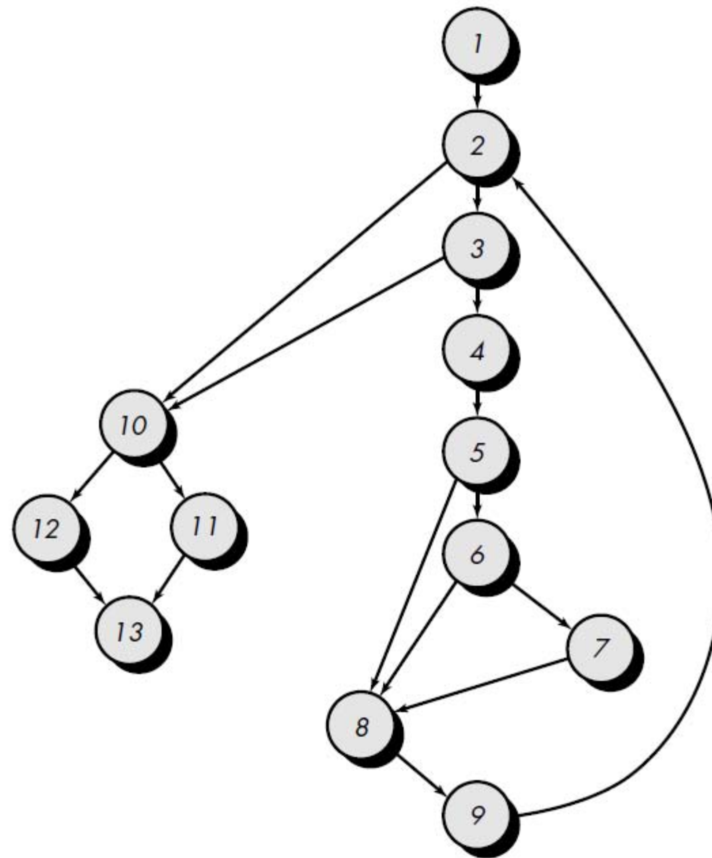
مسیر٤: ۱-۲-۳-٤-۵-۸-۹-۲-...

مسیر۵: ۱-۲-۳-٤-۵-۶-۸-۹-۲-...

مسیر۶: ۱-۲-۳-٤-۵-۶-۷-۸-۹-۲-...

سه نقطه ای که بعد از مسیرهای ٤،۵و ۶ می آید نشان می دهد که هر مسیر در باقیمانده ساختار کنترلی قابل توجه است غالبا خوب است در به دست آوردن موارد آزمون از گره های گزاره ای کمک گرفت. دراین مورد گروههای ۳،۵،۶،۱۰و۲ گروههای گزاره ای هستند.

FIGURE 17.5
Flow graph for the procedure **average**



شکل ۵-۱۷ گراف گردشی برای رویه average

٤.**موارد آزمونی را تهیه کنید که اجرای همه مسیرها در مجموعه پایه را الزامی کنند**. داده ها را باید طوری انتخاب کرد که به موازات آزموده شدن هر مسیر شرایط در گره ها ی گزاره ای بطور مناسب تنظیم شود موارد آزمونی که مجموعه پایه شرح داده شده در بالا را رعایت کنند عبارتند از:

مورد آزمون مسیر ۱:

ورودی معتبر، که در آن $k<i$ (i در زیر تعریف شده است)=value(k)

value(i)=-۹۹۹(که در آن)

نتایج مورد انتظار: میانگین صحیح بر اساس k مقدار و حاصل جمع مناسب.

توجه مسیر ۱ را نمی توان مستقلا آزمود بلکه باید آن را به عنوان بخشی از آزمونهای مسیر ٤و٥و ۶ آزمایش کرد.

مورد آزمون مسیر ۲:

value(1) = -۹۹۹

نتایج موردانتظار: ۹۹۹- =average؛ مقادیر حاصل جمع دیگر، مقادیر اولیه هستند.

مورد آزمون مسیر۳:

کوشش برای پردازش ۱۰۱ مقدار یا بیشتر.

۱۰۰مقدار نخست باید معتبر باشد.

نتایج مورد انتظار: همانند مورد آزمون ۱.

مورد آزمون مسیر ٤:

ورودی معتبر که درآن value(i)=i<۱۰۰

value(k)>minimum (k< iکه درآن)

نتایج مورد انتظار: میانگین صحیح براساس k مقدار و حاصل جمع مناسب.

مورد آزمون مسیر۵:

ورودی معتبر که درآن value(i)=i<۱۰۰

value(k)>maximum (k< iکه درآن)

نتایج مورد انتظار: میانگین صحیح براساس n مقدار و حاصل جمع کل.

مورد آزمون مسیر ۶:

ورودی معتبر که در آن value(i)=i<۱۰۰

نتایج مورد انتظار:میانگین صحیح بر اساس n مقدار و حاصل جمع کل.

هر مورد آزمون اجرا می شود و با نتایج مورد انتظار مقایسه می شود هنگامی که همه موارد آزمون کامل شدند آزمونگر می تواند مطمئن شود که همه دستورهای برنامه حداقل یک بار اجرا شده اند.

لازم به ذکر است که برخی مسیرهای مستقل (مسیر ۱درمثال ما) را نمی توان به شیوه ای مستقل آزمود. یعنی تلفیق داده های لازم برای عبور از این مسیر را نمی توان در جریان عادی برنامه به دست آورد. درچنین مواردی این مسیرها را به عنوان بخشی از یک آزمون مسیر دیگر می توان مورد آزمایش قرار داد.

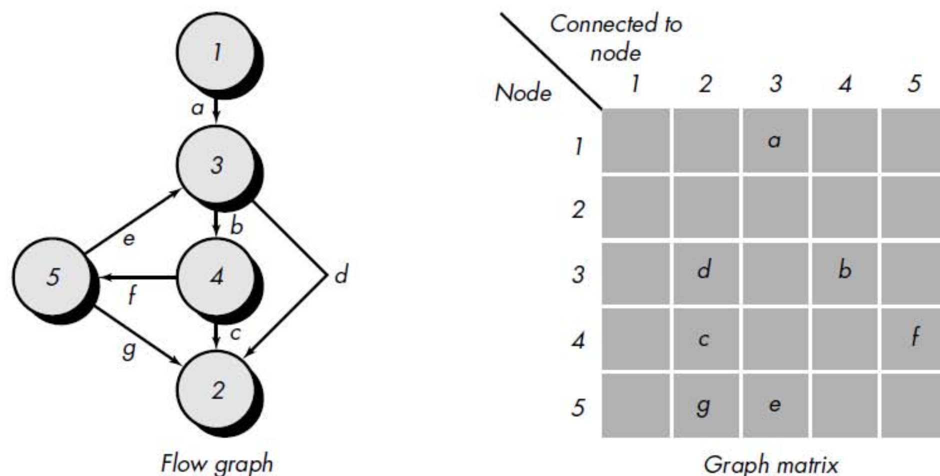
۴-۱۷-۴ ماتریس گراف

روش به دست آوردن گراف جریان و حتی تعیین مجموعه ای از مسیرهای پایه را می توان مکانیزه کرد توسعه یک ابزار نرم افزاری که به آزمون مسیرهای پایه کمک کند و ساختمان داده ای موسوم به ماتریس گراف می تواند بسیار مفید واقع شود.

ماتریس گراف یک ماتریس مربعی است که اندازه آن (یعنی تعداد سطر ها و ستونهای آن)برابر تعداد گره های موجود در گراف جریان است هر سطر و ستون متناظر با یکی از گره ها است و مدخلهای ماتریس با اتصالات (یعنی یالها)میان گره ها متناظرند. یک مثال ساده از گراف گردشی و ماتریس گراف متناظر با آن [BE190]BE190 در شکل ۶- ۱۷نشان داده شده است.

چنانکه از شکل پیداست هر گره از گراف جریان با یک شماره مشخص شده است حال آنکه هر یال با یک حرف الفبا. مدخل حرفی درماتریس برای نشان دادن ارتباط میان دو گره به کار رفته است برای مثال گره ۳ توسط یال b به گره ۴ متصل شده است.

تا اینجای کار، ماتریس گراف چیزی بیش از یک نمایش جدول بندی شده از گراف جریان نیست. ولی با افزودن وزن پیوند به هریک از مدخلهای ماتریس می توان آن را به ابزاری پر قدرت برای ارزیابی ساختار کنترلی برنامه در اثنای آزمون تبدیل کرد. وزن پیوند اطلاعاتی درباره جریان کنترل فراهم می آورد. وزن پیوند در ساده ترین شکل خود برابر ۱(وجود ارتباط)یا۰ (نبود ارتباط) است ولی خواص جالب دیگری را نیز می توان به اوزان پیوند نسبت داد.



شکل ۶-۱۷ ماتریس گراف

- احتمال اینکه یک پیوند (یال)اجرا شود؛
- زمان پردازش صرف شده برای طی کردن یک پیوند؛
- حافظه لازم برای طی کردن یک پیوند؛
- منابع لازم برای طی کردن یک پیوند؛

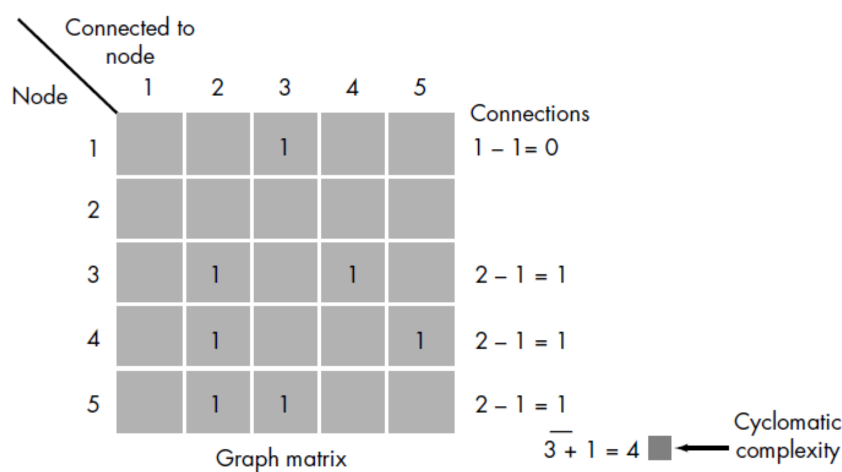
برای مثال از ساده ترین توزین برای نشان دادن ارتباطات استفاده می کنیم (۰یا۱). ماتریس گراف شکل ۶-۱۷ در شکل ۷-۱۷ دوباره رسم شده است. هر حرف جای خود را به عدد ۱ داده است که نشانگر وجود یک ارتباط است (برای وضوح بیشتر ۰ ها را نیاورده ایم).ماتریس گراف که به این شکل ارائه شده است ماتریس ارتباط نام دارد.

در شکل ۷-۱۷هر سطری با دو یا چند مدخل نشانگر یک گره گزاره ای است بنابراین با اجرای محاسبات نشان داده شده در طرف راست ماتریس ارتباط یک روش دیگر برای تعیین پیچیدگی سیکلوماتیک فراهم می آید(بخش۲-٤-۱۷).

بیزر[BE190] الگوریتمهای قابل اجرا روی ماتریسهای گراف را به طور کامل مورد بحث قرار داده است با استفاده از این تکنیک ها تحلیل لازم برای طراحی موارد آزمون را می توان به طور جزئی یا کامل خودکار کرد.

۵-۱۷آزمون ساختار کنترلی

تکنیک آزمون مسیرهای پایه یکی از چند تکنیک مربوط به آزمون ساختار کنترلی است. گرچه آزمون مسیر پایه ساده و بسیار کارآمد است به تنهایی کافی نیست دراین بخش اشکال دیگری از آزمون ساختار کنترلی رامورد بحث قرار می دهیم این اشکال کیفیت آزمون جعبه سفید را بهبود بخشیده پوشش دهی آن را وسعت می بخشند.



شکل ۷-۱۷ ماتریس ارتباط

۱-۵-۱۷آزمون شرط ها

آزمون شرط ها یک روش طراحی موارد آزمون است که شرطهای منطقی موجود در یک پیمانه برنامه را امتحان می کند هر شرط ساده، یک متغیر بولی یا یک عبارت رابطه ای است که ممکن است قبل از آ ن یک ("¬ ")NOT وجود داشته باشد عبارت رابطه ای به شکل زیر است:

E_1< عملگر رابطه ای >E_2

که درآن E_1 و E_2عبارت های محاسباتی و < عملگر رابطه ای > یکی از موارد "<"، ">" ، "="، "≠"، "≥"، "≤" است شرط مرکب از دو یا چند شرط ساده، عملگرهای بولی و پرانتز تشکیل می شود. فرض می کنیم که عملرهای بولی مجاز در یک شرط مرکب شامل OR(I)، AND("&")، NOT ("¬") باشد شرط بدون عبارتهای رابطه ای را عبارت بولی می گویند.

بنابرین انواع ممکن برای عناصر در یک شرط عبارتند از: عملگر بولی؛ متغیر بولی؛ یک جفت پرانتز بولی (که یک شرط ساده یا مرکب را محصور می کنند)؛ عملگر رابطه ای؛ یا یک عبارت محاسباتی.

اگر شرطی نادرست باشد در آن صورت حداقل یک جزءِ از شرط نادرست است، از اینرو انواع خطاها در یک شرط شامل موارد زیر می شود:

- خطای عملگر بولی(نادرست/جا افتاده/اضافی)
- خطای متغیر بولی
- خطای پرانتزهای بولی
- خطای عبارات محاسباتی

روش آزمون شرطها بر آزمون همه شرطهای موجود در برنامه تکیه دارد راهبردهای آزمون شرطها (که بعدا در همین بخش مورد بحث قرار خواهند گرفت) دومزیت دارند. نخست اندازه گیری وسعت پوشش دهی آزمون یک شرط ساده است دوم پوشش دهی آزمون یک شرط در برنامه راهنمایی برای تولید آزمونهای اضافی برای برنامه ارائه می کند.

هدف آزمون شرطها نه تنها یافتن خطاهای موجود در شرط های یک برنامه بلکه یافتن خطاهای دیگر موجود در برنامه است اگر یک مجموعه آزمون برنامه P در یافتن خطاهای موجود در شرطهای برنامه P موثر واقع شود این

P نیز موثر واقع شود به علاوه اگر یک راهبرد

آزمون برای یافتن خطاهای موجود در یک شرط موثر واقع شود این احتمال وجود دارد که آن راهبرد آزمون برای یافتن خطاهای موجود در برنامه نیز موثر واقع شود.

چند راهبرد برای آزمودن شرطها پیشنهاد شده است شاید آزمون شاخه ای ساده ترین راهبردها باشد. برای شرط مرکب C، شاخه های درست و نادرست C و هر شرط ساده در C باید حداقل یکبار اجرا شود [MYE79].

آزمون دامنه [WHI80] مستلزم به دست آوردن سه یا چهار آزمون برای یک عبارت گویا است برای یک عبارت گویا به شکل زیر:

E_1< عملگر رابطه ای >E_2

سه آزمون لازم است تا مقدار E_1 بزرگتر از، مساوی با، یا کوچکتر از E_2 باشد [HOW82]. اگر<عملگر رابطه ای > نادرست باشد و E_1وE_2 درست باشند در آن صورت با سه آزمون می توان اطمینان یافت که خطای عملگر رابطه ای پیدا خواهد شد. برای یافتن خطاها در E_1و E_2آزمونی که مقدار E_1را بزرگتر یا کوچکتر از E_2می کند باید بین این دو مقدار حداقل اختلاف ممکن را ایجاد کند.

برای یک عبارت بولی با n متغیر هر2n آزمون ممکن لازم است انجام شود (n>۰) با این راهبرد می توان خطاهای عملگر بولی متغیر و پرانتزها را یافت، ولی فقط در صورتی عملی است که n کوچک باشد.

برای عبارتهای بولی، آزمونهای حساس به خطا نیز می توان انجام داد [FOS84,TAI87]. برای هرعبارت بولی منفرد (یعنی عبارت بولی که درآن هر متغیر بولی تنها یک بار ظاهرشود) با N متغیر بولی (n>۰) می توان یک مجموعه آزمون با کمتر از 2N آزمون تولید کرد به طوری که این مجموعه آزمون یافتن خطاهای عملگرهای بولی چندگانه را تضمین کرده برای یافتن خطاهای دیگر موثر واقع شوند.

تای[TAI89] یک راهبرد برای آزمون شرطها پیشنهاد می کند که بر تکنیکهای فوق الذکر استوار است این تکنیک که به اختصار BRO خوانده می شود پیدا شدن خطاهای شاخه ای و عملگر رابطه ای را در یک شرط تضمین می کند مشروط بر اینکه همه متغیرهای بولی و عملگرهای رابطه ای فقط یک بار در شرط ظاهر می شوند و متغیر مشترکی نداشته باشند.

راهبرد BRO از محدودیتهای شرطی برا ی شرط C استفاده می کند محدودیت شرطی برای C با n شرط ساده به صورت (D_1,D_2,.....,D_n) تعریف می شود که ($D_i(0< i\leq n$ نماد مشخص کننده محدودیت روی نتیجه شرط ساده iام درشرط C است اگر طی اجرای C نتیجه هر شرط ساده C محدودیت مربوطه رادر D برآورد کند گفته می شودکه محدودیت شرطی D توسط C پوشش داده می شود.

برای متغیر بولی B یک محدودیت روی نتیجه B مشخص می کنیم که بیان می کند B باید درست (t) یا نادرست (f) باشد به طور مشابه در یک عبارت رابطه ای برای مشخص کردن محدودیتهایی روی نتایج عبارت از نمادهای >، =، < استفاده می شود. به عنوان مثال شرط زیر را در نظر بگیرید:

$$C_1:B_1\&B_2$$

که درآن B_1 و B_2متغیرهای بولی هستند محدودیت شرطی C_1 به شکل (D_1,D_2) است که درآن D_1 یا D_2 "t"یا "f" است. مقدار (t,f) یک محدودیت شرطی برای C_1 بوده توسط آزمونی که مقدار B_1 را درست و مقدار B_2را نادرست می کند پوشش داده می شود. راهبرد آزمون BRO مستلزم آن است که محدودیتهای {(t,t),(f,t),(t,f)} توسط اجرای C_1 پوشش داده می شود اگر C_1 به خاطر یک یا چند خطا در عملگرهای بولی نادرست باشد حداقل یک مجموعه از محدودیتها باعث شکست C_1 می شود.

به عنوان مثال دوم، شرطی به شکل زیررا در نظرمی گیریم:

$$C_1: B_1\&(E_3=E_4)$$

که در آن B_1 یک عبارت بولی و E_3و E_4 عبارتهای محاسباتی اند یک محدودیت شرطی برای C_2 به شکل (D_1,D_2) است که در آن D_1 یا "t" یا "f" و D_2، >، =، < است چون C_2 همانند C_1است. با این تفاوت که شرط ساده دوم در C_2 یک عبارت رابطه ای است، می توانیم یک مجموعه محدودیت برایC_2 در نظر بگیریم برای این منظور، مجموعه محدودیت های {(t,t),(f,t),(t,f)} را برای C_1 تعریف شد، اصلاح می کنیم. توجه کنید که "t"برای ($E_3=E_4$) به معنای "=" و f بر ($E_3=E_4$) به معنی "<" یا ">" است. با قرار دادن (t,=)و (f,=) به ترتیب به جای (t,t) و (f,t) و با قرار دادن (t,<) و (t,>) به جای (t,f)، مجموعه محدودیتهای شرطی برای C_2 عبارت است از: {(t,=), (f,=), (t,<), (t,>)} پوشش دهی این مجموعه، متضمن پیدا شدن خطاهای بولی و عملگر رابطه ای در C_2است.

به عنوان سومین مثال، شرطی به شکل زیر را در نظرمی گیریم:

$$C_3: (E_1>E_2)\&(E_3=E_4)$$

که درآن E_1، E_2 ، E_3، E_4 همگی عبارتهای محاسباتی هستند یک محدودیت شرطی برای C_3 به شکل (D_1,D_2) است که درآن D_1 و D_2هر یک >، =، < است. چون C_3همانند C_2است. با این تفاوت که شرط ساده در C_3 یک عبارت رابطه ای است، می توانیم یک مجموعه محدودیت برایC_3 به صورت زیر بنا کنیم:

$$\{(>,=),(=,=),(<,=),(>,>),(>,<)\}$$

پوشش دهی مجموعه محدودیتهای بالا تضمینی برای یافتن خطاهای رابطه ای در C_3است.

۲-۵-۱۷آزمون جریان داده ها

در روش آزمون جریان داده ها، مسیرهای آزمون یک برنامه، طبق موقعیت تعاریف و کاربردهای متغیرها در برنامه انتخاب می شود. برای آزمون جریان داده ها، چند راهبرد مورد مطالعه و مقایسه قرار گرفته است (مثل [FRA88]،[NTA88]،[FRA93]).

برای نشان دادن روش آزمون جریان داده ها فرض کنید به هر دستور از برنامه یک شماره دستور منحصر بفرد اختصاص داده شود و هیچ تابعی پارامترها یا متغیرهای سراسری خود را اصلاح نمی کند. برای دستوری با شماره دستور S.

DEF(S)={ X | دستور Sحاوی تعریفی از X باشد}

USE(S)={ X | دستور S حاوی کاربردی از X باشد}

اگردستور S یک دستور if یا حلقه باشد DEF آن خالی است و مجموعه USE آن مبتنی بر شرط دستور S است گفته می شود تعریف متغیر X در دستور S در دستور 'S زنده است اگر مسیری ازدستور S در دستور'S وجود داشته باشد که حاوی هیچ تعریف دیگری از X نباشد.

یک زنجیره تعریف کاربرد (یا زنجیره DU) از متغیر X به شکل[X,S,S'] که در آن S,S' شماره دستورها، X در DEF(S)و USE(S')است و تعریف X در دستور S در دستور'S زنده است.

یک راهبرد ساده برای آزمون جریان داده ها مستلزم آن است که هر زنجیره DU حداقل یکبار پوشش داده شود این راهبرد را راهبرد آزمون DU می نامند. ثابت شده است که آزمون DU پوشش دهی کلیه شاخه های برنامه را تضمین نمی کند. ولی تضمینی نیست که یک شاخه توسط آزمون DU پوشش داده شود، مگر در شرایط نادری مثل

if-then-else که درآنها بخش then هیچ متغیری را تعریف نمی کند و بخش else وجود ندارد در این وضعیت، شاخه else ازاین دستور if الزاما توسط آزمون DU پوشش داده نمی شود.

راهبردهای آزمون جریان داده ها برای انتخاب مسیرهای آزمون در برنامه های حاوی دستورهای حلقه و if تو در تو مفید هستند. برای نشان دادن این نکته، کاربرد آزمون DUبرای انتخاب مسیرهای آزمون مربوط به PDL زیر را درنظربگیرید:

```
proc x
   B1;
   do while C1
      if C2
         then
            if C4
               then B4;
               else B5;
            endif;
         else
            if C3
               then B2;
               else B3;
            endif;
         endif;
      enddo;
      B6;
   end proc;
```

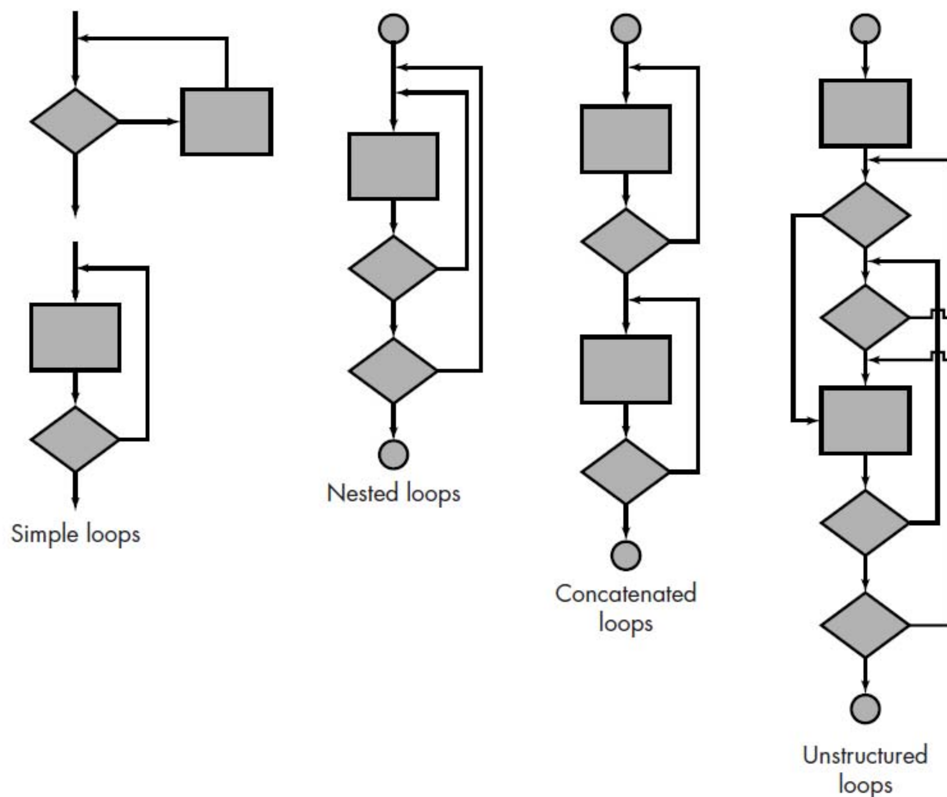
برای اجرای راهبرد آزمون DU جهت انتخاب مسیرهای آزمون نمودار جریان کنترل باید تعاریف و کاربردهای متغیرها را در هر شرط یا بلوک از PDL بدانیم. فرض کنید که X در آخرین دستور از بلوکهای B_5, B_4, B_2,B_3 , B_1 تعریف شود و در نخستین دستور از بلوکهای B_2,B_3,B_4,B5,B_6 به کار برده شود راهبرد آزمون DU نیازمند اجرای کوتاهترین مسیر از هر B_i $(0 < i < 5)$ به هر یک از B_j ها $(1 < B_j < 6)$ است (چنین آزمونی هر گونه کاربرد متغیرها x در شرایط C4,C3,C2,C1 را نیز پوشش می دهد) گرچه ۲۵ زنجیره DU از متغیر X وجود دارد فقط به ۵ مسیر نیاز داریم تا این زنجیره های DU را پوشش دهیم. علت آن است که برای پوشش دادن زنجیره DU مربوط به X از B_i $(0 < i \leq 5)$، تا B6 به ۵ مسیر نیاز است و زنجیره DU دیگر رامی توان با قرار دادن حلقه های تکرار در این ۵مسیر پوشش داد.

چون دستورات برنامه طبق تعاریف و کاربردهای متغیرها باهم ارتباط دارند روش آزمون جریان داده ها برای یافتن خطاها موثر واقع می شود. ولی مسائل اندازه گیری و پوشش دهی آزمون و انتخاب مسیرهای آزمون برای جریان داده ها از مسائل همتای خود در آزمون شرطها دشوارترند.

۳-۵-۱۷ آزمون حلقه ها

حلقه ها عناصر مهمی در الگوریتمهای نرم افزاری اند با این حال هنگام اجرای آزمونهای نرم افزاری توجه چندانی به آنها نمی شود.

آزمون حلقه ها یکی از تکنیکهای آزمون جعبه سفید است که انحصارا بر اعتبار ساختمان حلقه تکیه دارد چهار طبقه متفاوت از حلقه ها را می توان تعریف کرد [BEI90]: حلقه های ساده، حلقه های تسلسلی، حلقه های تودرتو، حلقه های ساخت نیافته (شکل ۸-۱۷).



شکل ۸-۱۷ انواع حلقه ها

حلقه های ساده. آزمونهای زیر را می توان درمورد یک حلقه ساده اجرا کرد، که درآن n حداکثر تعداد گذرهای مجاز از میان حلقه است.

۱. عدم اجرای حلقه
۲. فقط یک بار گذر ازحلقه
۳. دو بار گذر از حلقه
٤. m بار گذر ازحلقه که $m < n$ است
٥. n-1، n و n+1 گذر از حلقه

حلقه های تودرتو. اگر قرار بود روش آزمون مربوط به حلقه های ساده را به حلقه های تودرتو بسط دهیم تعداد آزمونهای ممکن با افزایش سطح تودرتویی به طور هندسی رشد می کرد. بیزر[BEI90] روشی پیشنهاد می کند که به کاهش دادن تعداد آزمونها کمک می کند:

۱. از داخلی ترین حلقه شروع کنید همه حلقه های دیگر را در حداقل مقدار قرار دهید.
۲. آزمونهای حلقه ساده را برای داخلی ترین حلقه اجرا کنید و درعین حال حلقه های خارجی دیگر را در حداقل مقدار پارامتر تکرارشان نگه دارید. آزمونهای دیگری برای مقادیر خارج از محدوده یا مقادیر مستثنی شده اجرا نمایید.
۳. به سمت بیرون حرکت کنید و همین روند را برای حلقه بعدی تکرار کنید، ولی همه حلقه های بیرونی را در حداقل مقدارشان و حلقه های داخلی را در مقادیر «معمولی» آنها قرار دهید.
٤. کار را تا آزمون همه حلقه ها ادامه دهید.

حلقه های تسلسلی.حلقه های تسلسلی را می توان با استفاده از روش آزمون برای حلقه های ساده آزمود. با این شرط که هریک از حلقه ها مستقل از دیگری باشد. ولی اگر دو حلقه تسلسلی نباشند و شمارنده حلقه ۱ به عنوان مقدار اولیه ای برای حلقه ۲ استفاده شود در آن صورت حلقه ها مستقل از یکدیگر نیستند. در این حالت روش به کار رفته در حلقه های تودرتو را توصیه می کنیم.

حلقه های ساخت نیافته. درصورت امکان این نوع از حلقه ها را باید طوری دوباره طراحی کرد که منعکس کننده کاربرد ساختمانهای برنامه نویسی ساخت یافته باشند.

۶-۱۷آزمون جعبه سیاه

آزمون جعبه سیاه که آزمون رفتاری نیز خوانده می شود بر خواسته های عملیاتی نرم افزار تکیه دارد. یعنی آزمون جعبه سیاه مهندس نرم افزار را قادر می سازد تا مجموعه هایی از شرطهای ورودی را به دست آورد که همه خواسته های عملیاتی برنامه را به طور کامل امتحان کند. آزمون جعبه سیاه جایگزینی برای تکنیکهای جعبه سفید به شمار نمی رود بلکه یک روش مکمل است که احتمال پیداکردن دسته دیگری از خطاها رافراهم می آورد.

آزمون جعبه سیاه سعی می کند خطاهای موجود در این گروهها را بیابد:۱.عملکرد نادرست یا جا افتاده؛ ۲.خطاهای واسط؛ ۳.خطاهای موجود در ساختمان داده ها یا دستیابی به بانک اطلاعاتی خارجی؛ ٤.خطاهای رفتاری یا کارایی و ۵.خطاهای مقدار دهی اولیه یا خاتمه برنامه.

بر خلاف آزمون جعبه سفید که از اوایل فرایند آزمون اجرا می شود آزمون جعبه سیاه را باید در مراحل آخر آزمون به کار برد چون آزمون جعبه سیاه ساختار کنترلی را در نظر نمی گیرد توجه به دامنه اطلاعاتی معطوف می شود برای پاسخ دادن به پرسشهای زیر آزمونهایی طراحی شده است.

- اعتبار عملیاتی چگونه آزموده می شود؟
- رفتار و کارایی سیستم چگونه آزمایش می شود؟
- چه طبقه ای از ورودیها موارد آزمون خوبی می سازند؟
- آیا سیستم نسبت به بعضی از ورودیها حساسیت دارد؟
- مرزهای یک طبقه از داده ها چگونه معین می شود؟
- چه حجم و میزانی جریان از داده ها را سیستم می تواند تحمل کند؟
- ترکیبات مشخصی از داده ها چه تاثیری بر عملکرد سیستم دارند؟

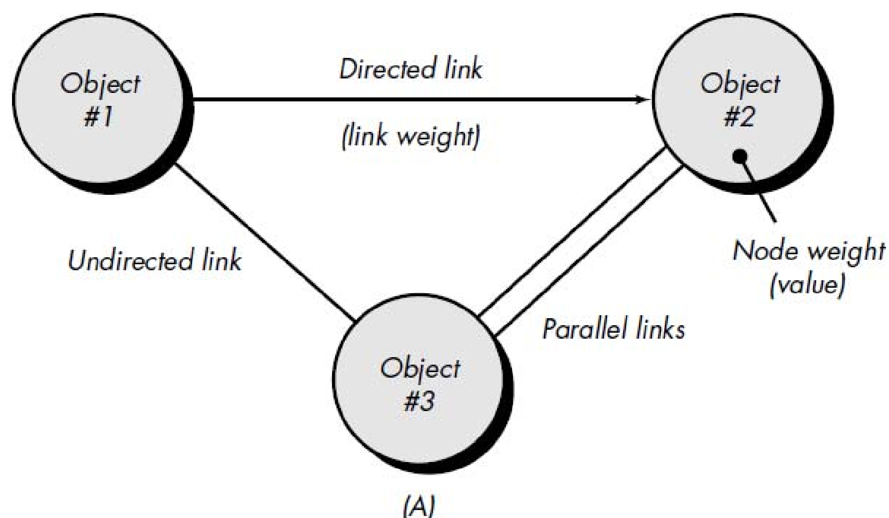
با اجرای تکنیکهای جعبه سیاه مجموعه ای از موارد آزمون به دست می آید که ملاکهای زیر را برآورده می کنند [MYE79]: ۱.موارد آزمونی که تعداد موارد آزمونی را که باید برای دستیابی به آزمون منطقی طراحی شوند، بیش از یک واحد کاهش می دهند و ۲.موارد آزمونی که درباره وجود یا نبود انواع خطاها اطلا عاتی به ما می دهند نه خطای مربوط به یک آزمون خاص.

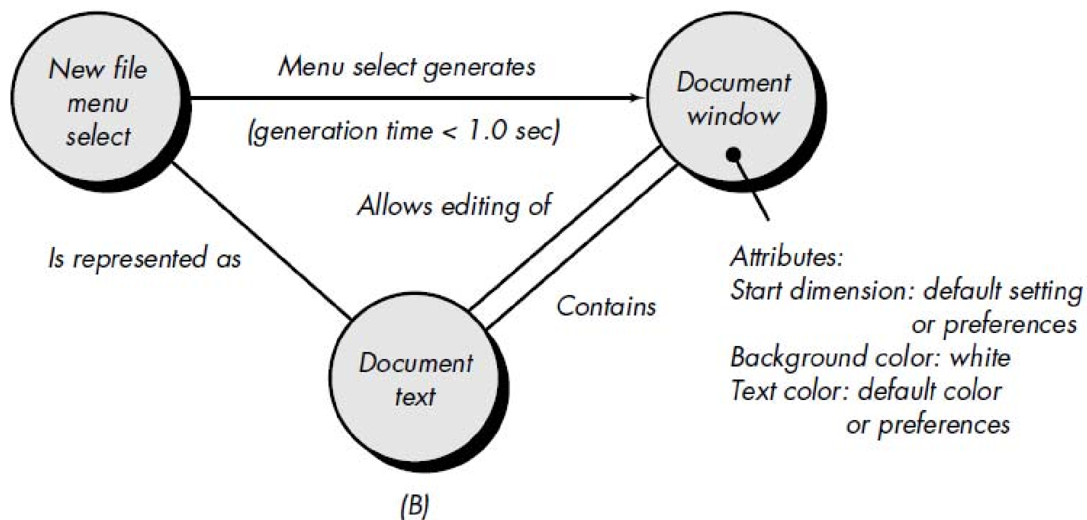
۱-۶-۱۷روشهای آزمون مبتنی بر گراف

نخستین گام در آزمون جعبه سیاه شناخت اشیایی است که در نرم افزار مدلسازی می شوند و نیز روابط میان این اشیا است. هنگامی که این منظور برآورده شد مرحله بعدی تعیین تعدادی آزمون است که ثابت کنند همه اشیاءِ دارای رابطه مورد انتظار را با یکدیگر دارند [BEI90]. به بیان دیگر آزمون نرم افزار با ایجاد گرافی از اشیای مهم و روابط میان آنها و سپس پی ریزی تعدادی آزمونها شروع می شود که گراف را پوشش می دهند. به طوری که هر یک از اشیا و روابط آن با اشیاءِ دیگری مورد آزمایش قرار می گیرد و خطاها کشف می شوند.

برای انجام این مراحل، مهندس نرم افزار با ایجاد یک گراف شروع می کند: مجموعه ای از گره ها که اشیاءِ را نشان می دهند؛ پیوندها که روابط میان اشیاءِ را نشان می دهند وزن گره که خواص گره را توصیف می کنند (مثلا مقدار داده ها یخاص یا رفتارحالت) و وزن پیوند که خواص پیوند را توصیف می کنند.

تصویری از یک گراف در شکل ۹-۱۷-الف نشان داده شده است گره ها به صوت دوایر نشان داده شده اند که توسط پیوندهای به هم متصل شده اند و این پیوندها اشکال متفاوتی به خود می گیرند. پیوند جهت دار (که با یک پیکان نشان داده می شود) نشان می دهد که رابطه تنها در یک جهت برقرار است. پیوند دو جهته که پیوند متقارن نیز خواند ه می شود بدان معنا است که رابطه در دو جهت برقرار است. از پیوندهای موازی زمانی استفاده می شود که چند رابطه متفاوت بین گره های گراف برقرار باشد.





شکل ۹-۱۷ الف نشانه گذاری گرافی ب یک مثال ساده

به عنوان یک مثال ساده، بخشی از گراف مربوط به یک برنامه واژه پرداز را در نظر بگیرید (شکل ۹-۱۷-ب)که درآن:

شی ء۱: **انتخاب فایل جدید از منو**

شیء۲: **پنجره سند**

شیء۳: **متن سند**

همان طور که از شکل پیداست با انتخاب گزینه فایل جدید یک پنجره سند تولید می شود. وزن گره پنجره سند لیستی از صفات پنجره را فراهم می آورد که انتظار می رود هنگام تولید پنجره موجود باشند. وزن پیوند نشان می دهد که پنجره باید در کمتر از ۱/۰ثانیه تولید شود. یک پیوند بدون جهت رابطه ای متقارن بین انتخاب منوی فایل جدید و متن سند برقرار می کند و پیوندهای موازی نشان دهنده روابط میان پنجره سند ومتن سند هستند. در حالت واقعی گراف تفضیلی تری باید برای طراحی مورد آزمون تهیه شود سپس مهندس نرم افزار موارد آزمون را با طی گراف و پوشش دادن کلیه روابط نشان داده شده به دست می آورد این موارد آزمون برای یافتن خطاهای موجود درهر یک از روابط طراحی می شوند.

بیزر [BEI90] چند روش آزمون رفتاری را توصیف می کند که در آنها از گرافها استفاده می شود:

مدلسازی جریان تراکنش. گره ها نشانگر مراحلی از یک تراکنش (مثلا مراحل لازم برای انجام رزرو بلیط هواپیما با استفاده از یک سرویس on-line) هستند و پیوندها ارتباط منطقی میان مراحل را نشان می دهند (مثلا بعد از flight.information.input، پردازش /قابلیت دسترسی /اعتبار سنجی می آید). برای ایجاد این گرافها می توان از نمودار جریان داده ها استفاده کرد.

مدلسازی حالت متناهی. گره ها نشانگر حالتهای مختلفی از نرم افزار هستند که توسط کاربر قابل مشاهده اند (مثلا هر یک از صفحات به محض گرفتن سفارش تلفنی توسط کارمند ظاهر می شوند) و پیوند، گذارهایی را نشان می دهند که برای حرکت از حالتی به حالت دیگر رخ می دهند (مثلا order-informationطی-inventory-availability look-up مورد تصدیق قرار می گیرد و سپس نوبت به customer-billing-information-input می رسد) برای ایجاد این نوع گرافها می توان از نمودار گذار حالت استفاده کرد.

مدلسازی جریان داده ها. گره ها اشیای داده هستند و پیوندها اتصالات تبدیلاتی هستند که برای تغییر دادن یک شیءِ به شیءِ دیگر رخ می دهند برای مثال، گره FICA.tax.withheld (FTW) با استفاده از رابطه FTW=0/62*GW از روی (GW)gross.wages محاسبه می شود.

مدلسازی زمان بندی. گره ها اشیای برنامه هستند و پیوند اتصالات ترتیبی میان آن اشیاءِ هستند وزن پیوند برای مشخص کردن زمانهای اجرای برنامه به کار می روند.

آزمون مبتنی بر گراف با تعریف کلیه گره ها و وزن آنها آغاز می شود یعنی اشیا و صفات آنها شناسایی می شوند مدل داده ها را می توان به عنوان نقطه شروع به کار برد ولی لازم به ذکر است که بسیاری از گره ها ممکن است اشیای برنامه ای باشند (به طور واضح درمدل داده ها نشان داده نشده باشند) برای بدست آوردن شاخصی از نقاط شروع و پایان گراف تعریف گره های ورودی و خروجی مفید واقع می شود.

هنگامی که گره ها شناسایی شدند پیوند ها و وزن پیوندها باید تعیین شود به طور کلی پیوندها را باید نامگذاری کرد. ولی پیوندهایی که نشانگر جریان کنترلی میان اشیای برنامه ای هستند نیازی به نامگذاری ندارند.

در بسیاری از موارد مدل گراف ممکن است حاوی حلقه هایی باشد (یعنی مسیری درگراف که یک یا چند گره در آن بیش ازیک بار بازدید شوند). آزمون حلقه ها (بخش ۳-۵-۱۷)را می توان درسطح رفتاری (جعبه سیاه) نیز اجرا کرد.

هر رابطه به طور جداگانه مطالعه می شود به طوری که موارد آزمون را می توان به دست آورد برای تعیین چگونگی تاثیر انتشار روابط میان اشیای تعریف شده در گراف از خاصیت تعدی روابط ترتیبی استفاده می شود. تعدی را با در نظر گرفتن سه شیءِ x،y وz می توان نشان داد. روابط زیر را در نظر بگیرید:

X برای محاسبه y لازم است.

Y برای محاسبه z لازم است.

بنابرین یک رابطه تعدی بین x وz برقرار شده است.

X برای محاسبه z لازم است.

براساس این رابطه تعدی آزمونهایی برای یافتن خطاها در محاسبه z با مقادیر گوناگونی را برای x وy در نظر بگیرند. تقارن یک رابطه (پیوند گراف) نیز راهنمایی مهم درطراحی موارد آزمون به شمار می رود. اگر پیوندی واقعا در جهتی متقارن) باشد آزمون این ویژگی اهمیت دارد ویژگی UNDO (لغو)[BEI90] در بسیاری از برنامه های کامپیوتری شخصی تقارن محدودی را پیاده سازی می کنند یعنی UNDO این امکان را فراهم می آورد که عملی پس از به انجام رسیدن لغو شود این ویژگی باید به طور کامل آزمایش شود و کلیه استثناها (یعنی جاهایی که UNDOقابل استفاده نیست) باید ذکر شود. سرانجام هر گره از گراف باید دارای رابطه ای باشد که به خودش منتهی شود؛ در اصل یک عمل بی اثر با عمل تهی. این روابط انعکاسی نیز باید آزموده شوند. همین که طراحی مورد آزمون شروع می شود هدف نخست پوشش دهی گره ها است یعنی آزمونها باید طوری طراحی شوند که مشخص کنند هیچ گرهی به طور ناخواسته حذف نشده است و وزن گره ها (صفات اشیاءِ) درست هستند.

سپس نوبت به پوشش دهی پیوندها می رسد برای مثال یک رابطه متقارن مورد آزمایش قرارمی گیرد تا نشان داده شود که دو جهتی است. یک رابطه تعدی مورد آزمایش قرارمی گیرد تا نشان داده شود که خاصیت تعدی وجود دارد یک رابطه انعکاسی آزمایش می شود تا اطمینان حاصل شود که یک حلقه تهی وجود دارد هنگامی که وزن پیوند ها مشخص شد آزمونهایی ابداع می شود تا نشان داده شود که این اوزان معتبرند سرانجام نوبت به آزمون حلقه ها می رسد.(بخش ۳-۵-۱۷).

۲-۶-۱۷ افراز هم ارزی

افراز هم ارزی یکی از روشهای آزمون جعبه سیاه است که دامنه ورودی یک برنامه را به طبقاتی از داده ها تقسیم می کند و موارد آزمون را می توان از روی آن به دست آورد یک مورد آزمون ایده ال یک طبقه از خطاها (مثلا پردازش نادرست همه داده های کاراکتری) را کشف می کند که در غیر این صورت قبل از مشاهده یک خطای عمومی موارد متعددی باید اجرا شوند. افراز هم ارزی مورد آزمونی را تعریف می کند که طبقاتی از خطاها را کشف می کند و در نتیجه از تعداد کل موارد آزمون می کاهد.

مورد آزمون برای افراز هم ارزی مبتنی برتعیین طبقات هم ارزی برای یک شرط ورودی است با استفاده از مفاهیمی که در بخش قبل معرفی شدند اگر بتوان مجموعه ای از اشیا را توسط روابط متقارن تعدی و انعکاسی به هم پیوند داد یک طبقه هم ارزی وجود دارد یک طبقه هم ارزی وجود دارد [BEI90]. طبق هم ارزی نشانگر مجموعه ای از حالتهای معتبر و نامعتبر برای شرایط ورودی است هر شرط ورودی معمولا با یک مقدار عددی بازه ای از مقادیر مجموعه ای از مقادیر مرتبط با یک شرط بولی است طبقات هم ارزی رامی توان طبق دستورالعملهای زیر تعریف نمود:

۱. اگر شرط ورودی، بازه ای رامشخص کند یک طبقه هم ارزی معتبر و دو طبقه هم ارزی نامعتبر تعریف می شوند.

۲. اگر شرط ورودی، نیازمند مقداری مشخص باشد یک طبقه هم ارزی معتبر، دو طبقه هم ارزی نامعتبر تعریف می شوند.

۳. اگر شرط ورودی، نیازمند عضوی از یک مجموعه باشد یک طبقه هم ارزی معتبر، یک طبقه هم ارزی نامعتبر تعریف می شوند.

٤. اگر شرط ورودی بولی باشد، یک طبقه معتبر ویک طبقه نامعتبر تعریف می شود.

به عنوان مثال داده هایی را در نظر بگیرید که به عنوان بخشی از یک برنامه بانکداری نگهداری می شوند. کاربرمی تواند با استفاده از کامپیوتر شخصی خودش به بانک دستیابی داشته باشد یک کلمه عبور شش رقمی را وارد کند و سپس عملیات بانکی را با تایپ فرمانها انجام دهد. طی دنباله ثبت رویدادها، نرم افزارها مربوط به داده ها را به شکل زیر می پذیرد:

کدناحیه –خالی یا ستاره سه رقمی

پیشوند –شماره ای سه رقمی که با ۰یا ۱شروع نمی شود.

پسوند-شماره ای چهار رقمی

کلمه عبور-رشته حرفی –عددی شش رقمی

فرمانها –(چک کردن)، (واریز کردن)، (پرداخت قبوض)و....

شرایط ورودی همراه با هر عنصر داده ای در برنامه بانکداری را می توان به صورت زیر مشخص کرد:

کدناحیه: شرط ورودی، بولی –کد ناحیه ممکن است وجود داشته باشد یا وجود نداشته باشد؛

شرط ورودی، بازه –مقادیر تعریف شده بین ۲۰۰و ۹۹۹ با استثناهای مشخص؛

پیشوند: شرط ورودی، بازه –مقدار مشخص شده بزرگتر از ۲۰۰ بدون ارقام صفر؛

شرط ورودی، مقدار –طول ٤رقم؛

کلمه عبور: شرط ورودی، بولی-کلمه عبور ممکن است وجود داشته باشد یا وجود نداشته باشد؛

شرط ورودی، مقدار-رشته شش کاراکتری

فرمان: شرط ورودی، مجموعه – حاوی فرمانهای فوق الذکر

با اجرای دستورالعملهای مربوط به نحوه به دست آوردن طبقات هم ارز، می توان موارد آزمونی برای هر قلم از داده های دامنه ورودی توسعه داد و اجرا نمود. موارد آزمون طوری انتخاب می شوند که بیشترین تعداد از صفات یک طبقه هم ارزی یکباره آزمایش شوند.

۳-۶-۱۷تحلیل مقادیر مرزی

به دلایلی که کاملا روشن نیست تعداد خطاهای موجود در مرزها دامنه ورودی نسبت به مقادیر مرکزی دامنه بیشتر است به همین دلیل تکنیکی موسوم به تحلیل مقادیر مرزی (BVA) توسعه یافته است تحلیل مقادیر مرزی به موارد آزمونی منجر می شود که مقادیر مرزی را امتحان می کنند.

تحلیل مقادیر مرزی یکی از تکنیکهای طراحی موارد آزمون است که مکمل افراز هم ارزی است BVA به جای انتخاب هر یک از عناصر طبقه هم ارزی به گزینش موارد آزمونی درلبه آن طبقه منجرمی شود و به جای آنکه فقط بر شرطهای ورودی تکیه کند موارد آزمونی از دامنه خروجی را نیزبه دست می آورد[MYE79].

دستورالعملهای BVA از بسیاری جهات مشابه با دستور العملهای ذکر شده و برای افراز هم ارزی است:

۱. اگر یک شرط ورودی بازه ای محصور به مقادیر aوb را مشخص کند موارد آزمونی با مقادیرaوb باید طراحی کرد که به ترتیب درست در پایین و بالای مقدار aوb باشند.
۲. اگر یک شرط ورودی چند مقدار را مشخص می کند باید موارد آزمونی توسعه یابند که اعداد بیشینه و کمینه را امتحان کنند مقادیری که درست در بالای بیشینه و درست در پایین کمینه قرار دارند نیز باید آزمایش شوند.
۳. دستورالعملهای ۱و۲ باید برای شرطهای خروجی نیز اعمال شوند فرض کنید که یک جدول دما در مقابل فشار به عنوان خروجی یک برنامه تحلیل مهندسی مورد نیاز است موارد آزمون باید طوری طراحی شوند که یک گزارش خروجی ایجاد کنند که حداکثر (وحداقل) تعداد مدخلهای ممکن برای جدول را مشخص کند.
٤. اگر ساختمان داده های داخلی برنامه دارای مرزهای معین باشند (مثلا آرایه ای دارای ۱۰۰مدخل باشد)حتما باید یک مورد آزمون برای امتحان کردن ساختمان داده طراحی شود.

اگر مهندسان نرم افزار به طور ضمنی BVA را تاحدی اجرا کنند با اعمال دستورالعملهای ذکر شده در بالا آزمون مرزی کامل می شود و در نتیجه احتمال پیدا شدن خطاها بازهم فزونی می یابد.

٤-۶-۱۷آزمون مقایسه ای

شرایط معینی وجود دارد (مثل کنترل هواپیما و کنترل نیروگاههای هسته ای) که در آنها قابلیت اطمینان نرم افزار اهمیت مطلق دارد. در چنین کاربردهای غالبا از نرم افزارها و سخت افزارهای اضافی برای به حداقل رساندن امکان خطا استفاده می شود هنگامی که نرم افزارهای اضافی توسعه می یابد تیمهای مهندسی نرم افزار جداگانه ای نسخه های مستقلی از برنامه های کاربردی را با استفاده از همان مشخصه ها توسعه می دهند. درچنین شرایطی هر نسخه را می توان با همان داده های آزمایشی مورد آزمون قرار داد تا اطمینان حاصل شود که همگی خروجی یکسان تولید می کنند سپس همه نسخه ها به طور موازی با مقایسه بلادرنگ نتا یج اجرا می شوند تا از سازگاری آنها اطمینان حاصل شود.

پژوهشگران با استفاده از درسهایی که از سیستمها فرا گرفته اند پیشنهاد کرده اند که برای کاربردهای بحرانی نسخه های مستقل از نرم افزار توسعه یابند حتی هنگامی که فقط یک نسخه در سیستم کامپیوتری تحویل شده مورد استفاده قرارمی گیرد. این نسخه های مستقل مبنای تکنیک آزمون جعبه سیاهی هستند که آزمون مقایسه یا آزمون پشت به پشت نامیده می شود.

هنگامی که یک مشخصه به چند صورت پیاده سازی شده باشد موارد آزمونی که با استفاده از تکنیکهای جعبه سیاه طرح شدند (مثلا افراز هم ارزی) برای همه نسخه های نرم افزار به عنوان ورودی در نظر گرفته می شوند اگر خروجیها یکسان باشند فرض می شود که همه پیاده سازی ها درست بوده اند اگر خروجی ها متفاوت باشند همه برنامه ها بررسی می شوند تا تعیین شود که آیا نقصی در یک یا چند نسخه منجر به این اختلاف شده است یا خیر. در اکثر موارد مقایسه خروجیها را می توان توسط یک ابزار خودکار انجام داد.

۵-۶-۱۷آرایه متعامد

دامنه ورودی بسیاری از برنامه های کاربردی محدود است یعنی تعداد پارامترهای ورودی کوچک و مقادیری که هریک از پارامترها به خود می گیرند دارای مرز مشخصی است. هنگامی که این اعداد بسیار کوچک باشند (مثلا ۳پارامترورود ی که هریک ۳مقدار مجزا می گیرند) می توان تمام حالتهای ورودی را در نظر گرفت و پردازش دامنه ورودی را به طور جامع مورد آزمایش قرار داد. ولی با رشد تعداد مقادیر ورودی و تعداد مقادیر مجزا جهت هر عنصر داده ای، آزمون جامع غیر عملی و امکان ناپذیر می شود.

آزمون آرایه متعامد. را می توان در مورد مسائلی به کار برد که در آنها ورودی نسبتا کوچک است ولی برای اجرای آزمون جامعیت بیش از حد بزرگ است روش آرایه متعامد در یافتن خطاهای مرتبط با خطاهای ناحیه ای مفید واقع می شوند-خطای ناحیه ای به گروهی از خطاها اطلاق می شود که منطق نادرست در نقطه ای از یک برنامه مربوط می شوند.

برای نشان دادن اختلاف میان آزمون آرایه متعامد و روش سنتی «یک عنصر ورودی درهر نوبت» سیستمی را در نظر بگیرید که دارای سه ورودی x،y و z است هر یک از این عناصر ورودی دارای سه مقدار هستند پس $۳^۳$=۲۷ مورد آزمون متفاوت امکان پذیر است. فادکه [PHA97] یک نمای هندسی از موارد آزمون متفاوت ارائه می دهد که در

هنگامی که آزمون آرایه متعامد رخ می دهد یک آرایه متعامد L9 از موارد آزمون تشکیل می شود. آرایه متعامد L9 یک خاصیت (متوازن کنندگی) است. [PHA97]یعنی موارد آزمون (که توسط نقاط سیاه در شکل نشان داده شده اند) به طور یکنواخت در سرتاسر دامنه آزمون پخش شده اند (مکعب سمت راست) پوشش دهی در دامنه ورودی کاملتر است.

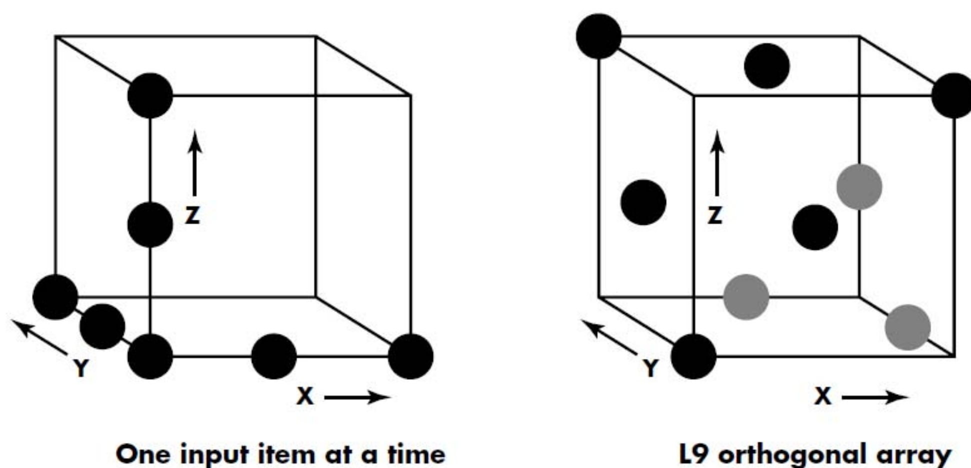
برای نشان دادن کاربرد آرایه متعامد L9 عمل ارسال را برای نمابر (fax) در نظر بگیرید چهار پارامتر p4,p3,p2,p1 به عمل ارسال تحویل داده می شود هر یک از آنها سه مقدار می گیرد مثلا p1 مقادیر زیر را می گیرد:

pl=1, همین الان ارسال کن

pl=2, یک ساعت بعد ارسال کن

pl=3,بعد از نیمه شب ارسال کن

p3،P2و p4 نیز مقادیر ۱،۲و۳ را به خود می گیرند که هر یک از حالتهای دیگری ارسال را نشان می دهند.



شکل ۱۰-۱۷ یک نمای هندسی از موارد آزمون

اگر از راهبرد آزمون (یک ورودی در هر نوبت) استفاده می شد، سری آزمونهای زیر(p4, p3, p2, p1) مشخص می شد: (۱،۱،۱،۱)،(۲،۱،۱،۱)،(۳،۱،۱،۱)،(۱،۲،۱،۱)،(۱،۳،۱،۱)،(۱،۱،۲،۱)،(۱،۱،۳،۱)،(۱،۱،۱،۲)،(۱،۱،۱،۳). فادکه [PHA97]این موارد آزمون را به شرح زیر ارزیابی می کند:

چنین موارد آموزشی فقط وقتی مفید واقع می شوند که شخص یقین دارد این پارامترهای آزمون با هم تعامل دارند آنها می توانند خطاهای منطقی را در جایی تشخیص دهند که یک پارامتر منفرد باعث بد کارکردن نرم افزار شود. این خطاها را خطاهای حالت یگانه می گویند. این روش قادر به یافتن خطاهای منطقی نیست که هنگام گرفتن مقادیر معینی از دو یا چند پارامتر به طور همزمان رخ می دهند. یعنی قادر به تشخیص تعاملها نیست. ازاین رو توانایی آن در یافتن خطاها محدود است.

نظر به تعداد نسبتا کوچک پارامترهای ورودی و مقادیر مجزا آزمون جامع امکان پذیر است. تعداد آزمون های لازم ۸۱ مورد است که گر چه بزرگ است، ولی عملی است. همه خطاهای مرتبط با حالتهای ترکیبی متفاوت داده ها پیدا خواهند شد ولی کار لازم برای رسیدن به این منظور نسبتا زیاد است.

روش آزمون آرایه متعامد در مقایسه با تست جامع با موارد آزمون کمتر پوشش خوبی را ارائه می کند آرایه متعامد l9 برای عمل ارسال نمابر در شکل ۱۱-۱۷ نشان داده شده است.

FIGURE 17.11 An L9 orthogonal array

Test case	Test parameters			
	P_1	P_2	P_3	P_4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

شکل ۱۱-۱۷ یک آرایه ارتوگونال L9

فادکه[PHA97] نتایج آزمون با استفاده از آرایه متعامد L9 را چنین ارزیابی می کند:

شناسایی و جداسازی کلیه خطاهای حالت یگانه. خطای حالت یگانه یک مشکل سازگار با هر یک از پارامترهای منفرد است. برای مثال اگر همه موارد آزمون با فاکتور P1=1 باعث ایجاد خطا شوند حالت یگانه شکست می خورد.

P=1) را به عنوان

منبع خطا جدا کرد یک چنین جداسازی خطایی برای رفع آن اهمیت دارد.

تشخیص کلیه خطاهای حالت دوگانه. اگر هنگام مقداردهی همزمان به دو یا چند پارامتر اشکالی ایجاد شود خطای حالت دوگانه وجود دارد در واقع خطای حالت دوگانه نشانی از ناسازگاری جفت شدگی یا تعامل زیانبار میان دو پارامتر آزمون است.

خطای چند حالتی. آرایه های متعامد (از نوع نشان داده شده)می توانند فقط یافتن خطاهای حالت یگانه و دوگانه را تضمین کنند ولی بسیاری از خطاهای چند حالتی نیز توسط این آزمونها قابل یافتن هستند.

۱۷-۷آزمون برای محیطها، معماریها و کاربردهای ویژه

با پیچیده تر شدن نرم افزارهای کامپیوتری نیاز به روشهای آزمون ویژه نیز فزونی یافته است روشهای جعبه سیاه و جعبه سفید که در بخش های ۵-۱۷و۶- ۱۷مورد بحث قرار گرفته اند در همه محیطهای، طراحی ها و کاربردها قابل اجرا هستند ولی گاه روشها و دستورالعملهای منحصر بفرد در مورد آزمون ضرورت پیدا می کند. در این بخش به دستورالعملهایی درباره آزمون در محیطها، معماریها و کاربردهای ویژه می پردازیم که معمولا مهندسان نرم افزار به آنها بر می خورند.

۱-۷-۱۷آزمون GUI ها

واسط گرافیکی کاربر(GUI) مشکلات جالبی سر راه مهندس نرم افزار می دهد. به خاطر وجود مولفه های قابل استفاده مجدد در محیطهای توسعه(GUI) ایجاد واسط کاربر با زمانی کمتر و با دقت بالاتر امکان پذیر شده است. ولی در عین حال پیچیدگی (GUI) نیز فزونی یافته است و در نتیجه طراحی و اجرای موارد آزمون دشوارتر شده است

از آنجا که بسیاری از GUIجدید دارای شکل و شمایل خاص هستند می توان به آزمونهای استاندارد دست یافت. گراف های مدل سازی حالتهای متناهی (بخش ۱-۶-۱۷-۶) را می توان برای بدست آوردن آزمونهایی به کار برد که با داده های مشخص و اشیای برنامه ای مرتبط با GUI سروکار دارند.

به خاطر تعداد زیاد حالتهای ممکن در عملیا ت GUI، آزمون باید با استفاده از ابزارهای خودکار انجام شود. طی چند سال اخیر انواع ابزارهای آزمون GUI به بازار ارائه شدند.

۲-۷-۱۷آزمون معماریهای مشتری/کارگزار

معماریهای مشتری/کارگزار (C/S) مشکل چشمگیری برای آزمونگر نرم افزار پیش می آورند. ماهیت توزیع شده محیطهای مشتری /کارگزار، مسائل کارایی مرتبط با پردازش تراکنشها حضور بالقوه چند سایت سخت افزاری متفاوت، پیچیدگیهای ارتباط شبکه ای، نیاز به متقاضیان چندگانه از یک بانک اطلاعاتی متمرکز (و در برخی موارد) توزیع شده و خواسته های هماهنگ سازی تحمیل شده به کارگزار همگی دست به دست هم داده آزمون معماریهای C/S و نرم افزارهای مربوط به آنها را دشوارتر از نرم افزارهای مستقل ساخته است درحقیقت مطالعات صنعتی جدید نشان دهنده افزایش چشمگیر زمان آزمون و هزینه در هنگام توسعه محیطهای C/S هستند.

۳-۷-۱۷ آزمون مستندات و تسهیلا ت راهنما

اصطلاح «آزمون نرم افزار» تعداد زیادی از موارد آزمون را در ذهن متبادر می کند که برای امتحان کردن برنامه کامپیوتری و داده هایی که دستکاری می کند تهیه شده اند با توجه به تعریف نرم افزار در فصل اول لازم به ذکر است که آزمون باید به سومین عنصر از پیکربندی نرم افزار یعنی مستندات توسعه یابد.

خطاهای موجود در مستندات برنامه می تواند به اندازه خطاهای موجود در داده ها یا کد منبع مخرب باشند. هیچ چیز ناراحت کنند ه تر از این نیست که دستورالعملهای موجود در جزوه راهنما یا راهنما ی on-line یک برنامه را دنبال کنید و نتایج پیش بینی شده را مشاهده نکنید. لذا آزمون مستندات باید در طراحی آزمون به حساب آورده شود. آزمون مستندات را در دو فاز می توان انجام داد در فاز نخست یعنی بازبینی و وارسی مستندات از لحاظ وضوح ویرایش مورد بررسی قرارمی گیرند. فاز دوم یعنی آزمون زنده از مستندات همراه با برنامه استفاده می شود.

آزمون زنده برای مستند سازی را می توان با استفاده از تکنیکهایی مشابه با روشهای آزمون جعبه سیاه (بخش ۶-۱۷) انجام داد از آزمون مبتنی بر گراف می توان برای توصیف کاربرد برنامه استفاده کرد؛ افراز هم ارزی و تحلیل مقدار مرزی را می توان برای تعیین طبقات گوناگونی از تعاملهای ورودی و مرتبط به کاربرد.

١٧-٧-٤آزمونهای مربوط به سیستم های بلادرنگ

ماهیت وابسته به زمان در بسیاری از کاربردهای بلادرنگ یک عنصر دشوار زمان را به آزمون می افزاید طراح موارد آزمون نه تنها باید موارد آزمون جعبه سیاه و جعبه سفید را در نظر بگیرد بلکه باید کنترل رویدادها (یعنی پردازش وقفه ها) زمانبندی داده ها و موازی بودن وظایفی (فرایندهایی) را که با داده ها کار می کنند نیز در نظر بگیرند در بسیاری از شرایط داده های آزمونی که در یکی از حالتهای سیستم بلادرنگ به دست آمده اند به پردازش مناسب منجر می شود در حالی که همان داده ها که درحالت دیگری از سیستم بلادرنگ به دست آمده اند ممکن است منجر به خطا شوند

برای مثال نرم افزارهای بلادرنگ که یک دستگاه فتوکپی جدید را کنترل می کند وقفه های اپراتور (یعنی وقتی اپراتور کلیدهای کنترلی مثل resetیا darken را می زند) را در حین گرفتن کپی (حالت کپی گرفتن) می پذیرد اگر دستگاه کپی در حالت گیر کردن کاغذ باشد و اپراتور کلیدهای کنترلی را فشار دهد ماشین دچار وقفه می شود و در نتیجه پیامی صادر می شود که مشخص می کند مکان گیرکردن کاغذ از دست رفته است (یک خطا).

به علاوه رابطه نزدیکی که بین نرم افزارهای بلادرنگ وجود دارد و محیط سخت افزاری آن نیز می تواند باعث ایجاد مشکلات در آزمون شود در آزمونهای نرم افزاری باید تاثیر اشکالا ت سخت افزاری بر پردازش نرم افزا ر نیز در نظر گرفته شود شبیه سازی واقعی چنین خطاهایی می تواند بی اندازه دشوار باشد

روشهای مفهومی طراحی موارد آزمون برای سیستمهای بلادرنگ هنوز باید تکامل پیدا کند ولی یک راهبرد چهار مرحله کلی می توان پیشنهاد کرد:

آزمون وظایف: نخستین گام در آزمون نرم افزار بلادرنگ این است که هر یک از وظایف به طور مستقل آزمایش شوند یعنی برای هر وظیفه آزمونهای جعبه سیاه و جعبه سفیدی طراحی و اجرا شوند طی این آزمونها، هر وظیفه به

آزمون رفتاری. با استفاده از مدلها ی سیستمی که توسط ابزارهای کیس ایجاد شدند می توان رفتار سیستم بلادرنگ را شبیه سازی کرد و رفتار آن را به عنوان پیامدی از رویدادهای خارجی بررسی کرد این فعالیتهای تحلیلی می توانند به عنوان مبنایی برای طراحی موارد آزمونی عمل کنند که هنگام ساخته شدن نرم افزار های بلادرنگ اجرا می شوند با استفاده از تکنیکی مشابه با افراز هم ارزی (بخش ۱-۶-۱۷) رویدادها (مثلا وقفه ها و سیگنالهای کنترلی)برای آزمون گروه بندی شده اند. برای مثال رویداد های مربوط به دستگاه فتوکپی ممکن است عبارت باشند از: وقفه های کاربر (مثل صفر کردن شمارنده)، وقفه های مکانیکی (مثل گیر کردن کاغذ)، وقفه های سیستمی (مثل کم شدن قدرت تونر) و حالتهای خطا (داغ شدن غلطک). هریک از این رویدادها به طور انفرادی مورد آزمون قرار می گیرد و رفتار سیستم اجرایی مورد بررسی قرار می گیرد تا خطاهایی که در نتیجه پردازش این رویدادها رخ می دهند برملا شوند. رفتار مدل سیستمی (که طی فعالیت تحلیل توسعه می یابد) و نرم افزار قابل اجرا را می توان برای همخوانی مورد مقایسه قرار داد. هنگامی که انواع رویدادها مورد آزمون قرار گرفتند رویدادها به ترتیب تصادفی و با فراوانی تصادفی به سیستم ارائه می شود. وظایف همزمان سازی بین وظایف رخ می دهد یا خیر. به علاوه وظایفی که از طریق صفی از پیامها یا انبار داده ها با هم ارتباط برقرار می کنند، آزمایش می شوند تا خطاهای موجود در تعیین اندازه نواحی نگهداری داده برملا شوند.

آزمون سیستم. نرم افزار و سخت افزار به هم گره خورده اند و گستره کاملی از آزمونها ی سیستم اجرا می شوند تا خطاهای موجود در واسط میان سخت افزار و نرم افزار پیدا شوند.

اکثر سیتمهای بلادرنگ وقفه ها را پردازش می کنند بنابراین آزمون مربوط به کنترل این رویدادهای بولی ضروری است آزمونگر با استفاده از نمودار گذار حالت و مشخصه کنترلی لیستی از همه وقفه های ممکن و پردازشی تهیه می کند که به عنوان پیامدی از وقفه رخ می دهد. سپس آزمونهایی طراحی می شوند که ویژگی سیستمی زیر را مورد ارزیابی قرار دهند

- آیا اولویتهای وقفه به طور مناسب منتسب وکنترل می شوند؟
- آیا پردازش مربوط به هر وقفه به درستی کنترل شده اند؟

- آیا کارایی(مثلا زمان پردازش) برای هر یک از رویدادها ی کنترل وقفه از خواسته ها تبعیت می کنند؟
- آیا حجم بالایی از وقفه ها که در زمانهای بحرانی رخ می دهند مشکلاتی را درعملکرد یا کارایی به وجود می آورند؟

علاوه بر این نواحی داده های سرتاسری که برای انتقال دادن داده ها به عنوان بخشی از پردازش وقفه به کار می روند باید مورد آزمون قرار گیرند تا توان بالقوه برای ایجاد اثرات جانبی سنجیده شود.

6644462