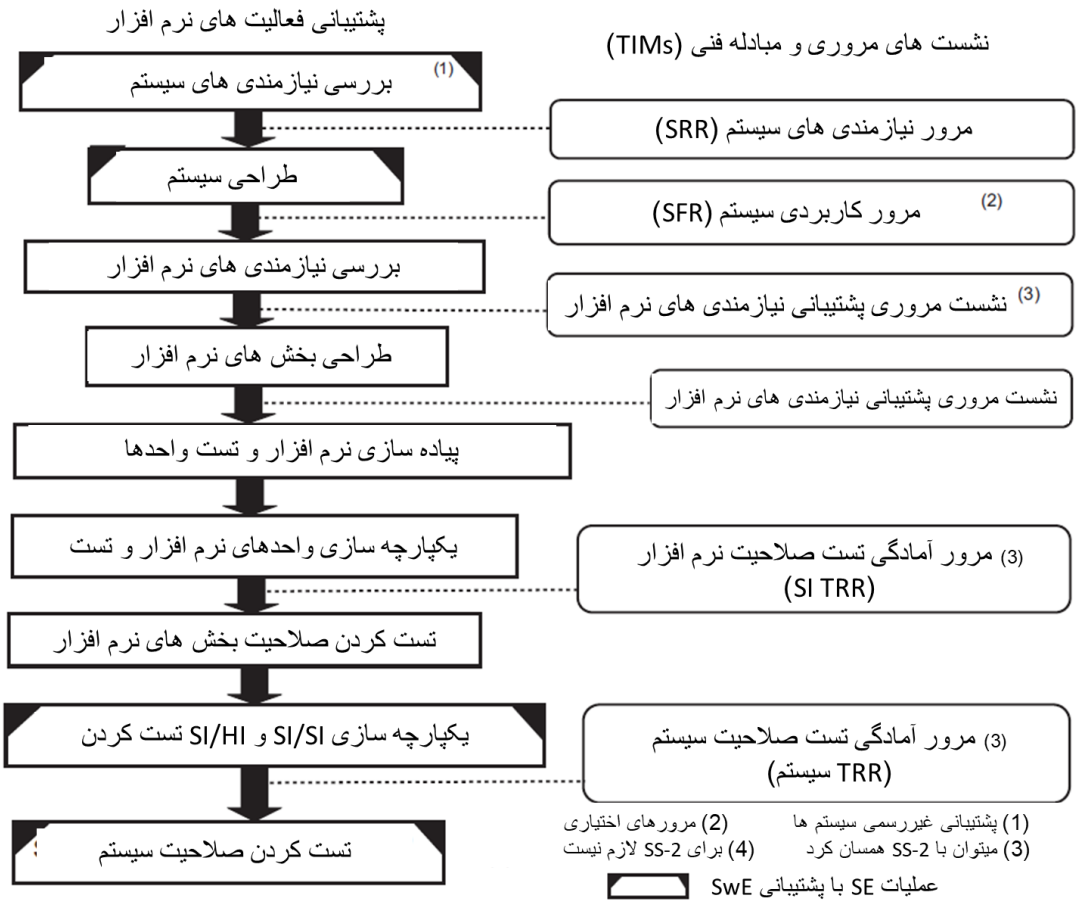
باسمه تعالی

ترجمه صفحات ۴۳ تا ۴۷ کتاب

Project management of large software-intensive systems : controlling the software development process

حانیه رضایی بقا ۹۸۰۰۰۰۵۲۴



شکل ۳-۷- فرایند توسعه نرم افزار پشتیبانی

۳-۹- فرایندهای تست نرم‌افزار

تست کردن اختیاری نیست؛ این فرآیند یک عنصر ذاتی و حیاتی در هر سیستم نرم‌افزاری است.

مدیریت فرآیند تست اساسا شامل برآورد هزینه تست است؛ پی‌گیری و نظارت بر هزینه واقعی آزمایش؛ و سعی برای اینکه که در مسیر خود بماند. من می‌توانم کتابی در مورد آزمایش بنویسم، بنابراین برای جلوگیری از ایجاد

یک کتاب راهنمای ۱۰۰۰ صفحه‌ای، هفت موضوع آزمون انتخاب شده‌اند تا به طور خلاصه در مورد چگونگی ارتباط آن‌ها با مدیریت فرآیند آزمون بحث شود:

- تست کردن یک استراتژی ظالمانه است. (۳-۹-۱)
- استراتژی های تست (۳-۹-۲)
- تست رگرسیون و فاکتوریل (۳-۹-۳)
- آزمون بررسی ریسک (۳-۹-۴)
- قابلیت آزمایش الزامات (۳-۹-۵)
- عملیات test-like-you (۳-۹-۶)
- زمان توقف تست (۳-۹-۷)

۳-۹-۱- تست کردن بسیار ظالمانه است.

تست‌های سیستم و توابع تست نرم‌افزار در زمان‌های مختلف، با سطوح مختلف وفاداری، در طول چرخه حیات توسعه نرم‌افزار انجام می‌شوند. در این زمینه، تست نرم‌افزار چندین بار به عنوان بخشی ازفعالیت‌های زیر، در مکان‌های مشخص‌شده در این کتاب اشاره شده است:

- برنامه ریزی تست مورد نرم افزار
- برنامه ریزی تست سیستم
- تست ، تأیید و اعتبارسنجی نرم افزار
- کدگذاری نرم افزار و تست واحد
- یکپارچه سازی و تست واحد نرم افزار
- تست صلاحیت مورد نرم افزار
- تست یکپارچه سازی مورد نرم افزار و مورد سخت افزار
- یکپارچه سازی و آزمایش زیر سیستم ها
- تست صلاحیت سیستم

۳-۹-۲- استراتژی‌های آزمون

به عنوان SPM، به ویژه در برنامه‌های بزرگ، شما نباید مستقیما تست واقعی را انجام دهید. با این حال، باید در تصمیم‌گیری درباره استراتژی تست مناسب برنامه‌ریزی‌شده برای پروژه خود مستقیما درگیر باشید. پرداختن به

استراتژی به معنای یک رویکرد کلی به جای روش‌ها و تکنیک‌های خاص طراحی‌شده برای آزمایش اجزا است. استراتژی‌های اصلی تست عبارتند از:

- تست آلفا و بتا: اغلب برای محصولات نرم‌افزاری تجاری قبل از انتشار رسمی برای شناسایی نقایص استفاده می‌شود. نسخه قبل از انتشار به گروه کوچکی از نمایندگان کاربران برای آزمایش داده می‌شود. تست آلفا معمولا داخلی است و تست بتا معمولا توسط کاربران خارجی انجام می‌شود.
- تست بازگشت به عقب: یک استراتژی تست زمانی استفاده می‌شود که نسخه‌های یک سیستم موجود باشند. نسخه‌ها با هم تست می‌شوند، و خروجی آن‌ها از نظر ثبات مقایسه می‌شود.
- تست پایین به بالا: تست با اجزای اساسی شروع می‌شود و به سمت بالا کار می‌کند.
- تست پیکربندی: تجزیه و تحلیل نرم‌افزار تحت پیکربندی‌های مختلف مشخص‌شده زمانی که نرم‌افزار برای خدمت به بیش از یک کاربر ساخته شده‌است.
- تست نصب: بررسی عملکرد نرم‌افزار در محیط هدف.
- تست عملکرد: تایید می‌کند که نرم‌افزار توسعه‌یافته نیازهای عملکردی مشخصی را برآورده می‌کند.
- تست بازیابی: نرم‌افزار نیروها به روش‌های مختلف شکست می‌خورد و اثبات می‌کند که بازیابی از شکست اجباری به درستی و زیبایی انجام می‌شود.
- آزمون رگرسیون: تایید می‌کند که اصلاحات باعث اثرات ناخواسته نشده اند (۳-۹-۳ را ببینید).
- آزمایش امنیتی: تایید می‌کند که مکانیسم‌های حفاظتی داخلی از دسترسی نامناسب جلوگیری خواهد کرد.
- آزمایش استرس: فشار بر روی سیستم را با فراتر رفتن از بار طراحی مشخص‌شده و محدودیت‌های طراحی آن تنظیم می‌کند، به عنوان مثال، چگونه سیستم می‌تواند با شرایط بار اضافی مقابله کند.
- به عنوان مثال برای سیستم‌های تست با فرآیندهای چندگانه استفاده می‌شود که در آن پردازش رشته‌های تراکنش در این فرآیندها راه خود را باز می‌کند.
- تست بالا - پایین: تست کردن با انتزاعی ترین جز شروع می‌شود و به سمت پایین کار می‌کند.

سیستم‌های بزرگ‌تر معمولا با استفاده از ترکیبی از این استراتژی‌ها تست می‌شوند تا هر رویکرد واحد. ممکن است استراتژی‌های مختلفی برای بخش‌های مختلف سیستم و در مراحل مختلف در فرآیند تست مورد نیاز باشد. توسعه نرم‌افزار بویژه در پروژه‌های بزرگ، کار کوچکی نیست، بنابراین شما باید زمان و منابع زیادی را برای تولید این سند مهم برنامه‌ریزی فراهم کنید.

درس‌های گرفته شده: استراتژی تست هر چه که اتخاذ شود، همیشه منطقی است که از یک رویکرد فزاینده برای سیستم فرعی و تست سیستم استفاده شود. همه اجزا را یکپارچه نکنید و سپس شروع به آزمایش کنید. همیشه سیستم را به صورت تدریجی تست کنید. هر افزایش باید قبل از اضافه شدن افزایش بعدی به سیستم تست شود. این فرآیند باید ادامه یابد تا تمام ماژول‌ها در سیستم گنجانده شوند. هنگامی که یک ماژول جدید در فرآیند تست معرفی می‌شود، تست‌های قبلی که هیچ نقصی پیدا نکرده اند، اکنون می‌توانند نقایص را تشخیص دهند. این نقایص احتمالا ناشی از تعامل با ماژول جدید است. منبع مشکل ممکن است محلی باشد تا محل نقص و ترمیم را ساده کند.

۳-۹-۳- آزمون رگرسیون و فاکتوریل

آزمون رگرسیون. این یک رویکرد آزمون مجدد است که برای تایید اینکه کد تغییر می‌کند مشکلات شناسایی‌شده را ثابت می‌کند، اما ارزش واقعی آن اثرات ناخواسته تغییرات کد را پیدا می‌کند. تست رگرسیون خوب باید قادر به شناسایی اثرات محلی و جهانی تغییرات کد باشد. پیدا کردن نقایصی که به میزان ۵ تا ۱۰ درصد از نقایص تزریق شده‌باشد، غیر عادی نیست. میزان خطاهای غیر عمدی ناشی از ایجاد تغییرات کد به شدت به پیچیدگی طراحی بستگی دارد زیرا پیچیدگی می‌تواند اصلاح عیوب نرم‌افزار را دشوارتر کند و اغلب عوارض جانبی آن‌ها کم‌تر آشکار است.

به عنوان SPM، ممکن است در مورد نیاز به انجام تست رگرسیون به دلیل تاثیر آن بر هزینه و زمانبندی دچار اختلاف‌نظر شوید. ممکن است مجبور شوید یک بحث قوی را آماده کنید به خصوص اگر مدیر ارشد شما هدف اصلی باشد. آمار را نگه دارید تا ثابت کنید که توسعه‌دهندگان شما خطاهای کافی را تزریق می‌کنند تا تست رگرسیون را توجیه کنند.

درس های گرفته شده: نتیجه نهایی این است که هر زمان که یک تغییر کد وجود دارد، آزمایش قبلی باید تکرار شود تا تایید شود که آن تغییرات رفتار نرم‌افزار قبلا تست شده را تغییر نمی‌دهد. به نظر من، اگر هر عنصر سیستمی شامل تغییرات سخت‌افزاری تغییر کند، تست نرم‌افزار باید تکرار شود.

یک نمونه معروف و گران‌قیمت از عدم پیروی از این توصیه، شکست وسیله نقلیه پرتاب موشک آریانه وی در سال ۱۹۶۶ بود. نرم‌افزار پرواز بدون نقص در راه‌اندازی قبلی آریان ۴ کار می‌کرد. این نرم‌افزار برای آریانه ۵ دوباره تست نشد چون هیچ تغییری در نرم‌افزار ایجاد نشد. با این حال، تغییراتی در سخت‌افزار ایجاد شد، و نرم‌افزار کنترلی که از آریانه ۴ استفاده می‌کرد در طول تست پرواز آریانه ۵ با شرایطی مواجه شد که دستورها اشتباه را درک و صادر نکرد و باعث شد که خودرو از کنترل خارج شود. آریانه پنجم باید نابود می‌شد.

مقدار شکست ناشی از اثرات ناخواسته تغییرات کد را می توان با کاهش منابع کاهش داد که می‌تواند باعث ایجاد نقایصی شود که به طور تصادفی تزریق می‌شوند. اطمینان حاصل کنید که کل تیم توسعه نرم‌افزار در فرآیند شناسایی این منابع مشارکت دارند. حتی با وجود فرایندهای توسعه نرم‌افزار بسیار خوب، برخی نواقص همچنان تزریق خواهند شد، به طوری که یک پروتکل آزمون برگشت اجباری باید بر پروژه - به ویژه برای پروژه‌های بزرگ‌تر - تحمیل شود.

فاکتوریل. اضافه کردن، حذف یا تغییر کد منبع به منظور ساده‌تر کردن خواندن، درک و نگهداری کد، فاکتوریل نامیده می‌شود. شناسایی با تغییر طراحی و رویکرد، بهبودهایی را ایجاد می‌کند، اما رفتار یا کارکرد نرم‌افزار را تغییر نمی‌دهد. بسیاری از برنامه نویسان فرصت‌هایی برای بهبود طراحی سیستم پیدا می‌کنند. اما این بهبودها معمولا تا زمانی که برنامه نویسان بتوانند کد را به خوبی توسعه دهند، برای آن‌ها آشکار نمی‌شود. تکنیک‌های بهبود مختلفی وجود دارند که می توان آن‌ها را فاکتور سازی مجدد نامید (آن‌ها در اینجا توصیف نخواهند شد).

ایجاد کارایی بیشتر در سیستم و آسان‌تر کردن درک و نگهداری کد، چیز خوبی است. بااینحال، صرف زمان بیش از حد برای بهتر و بهتر کردن آن می‌تواند برای هزینه پروژه و اهداف زمانبندی مخرب باشد. به عنوان مدیر پروژه نرم‌افزار، شما باید تصمیم بگیرید که چه زمانی خط بکشید و بعد از رفع نیازهای قراردادی تصمیم بگیرید که چه زمانی به اندازه کافی خوب است. برای بحث کامل در مورد فاکتور سازی به Fowler (۲۰۰۰) مراجعه کنید.

شما همیشه می‌توانید محصول را "بهتر" کنید، بنابراین اگر این تلاش برای مدت طولانی ادامه یابد بودجه شما را از بین خواهد برد. بحث آبکاری طلا را در بخش ۵. ۵ ببینید تا به پی‌گیری تاثیر تجدید ساختار طولانی کمک کنید. البته، اگر زندگی‌ها به عملکرد ثابت نرم‌افزار وابسته باشند، هدف، کمال یا تا حد امکان نزدیک به آن است.

۳-۹-۴- آزمون بررسی ریسک

از آنجا که تست کردن هر جنبه ممکن یک برنامه غیر ممکن است، هر ترکیب ممکن از وقایع، هر رابط وابستگی و هر چیز دیگری که می‌تواند اشتباه کند، تحلیل ریسک تست برای بیشتر پروژه‌های توسعه نرم‌افزار مناسب است. از آنالیز ریسک برای تعیین محل تمرکز تست استفاده کنید. این کار به قضاوت، عقل سلیم و تجربه نیاز دارد. از یک چک لیست شبیه به مثال جدول ۳-۲ برای به دست آوردن پاسخ به این نوع سوالات متناسب با برنامه خود استفاده کنید.

آزمایش معمول انجام‌شده برای نرم‌افزارهای مصرف‌کننده برای دستگاه‌های پزشکی کافی نیست. الزامات آزمایش دقیق دستگاه‌های حیاتی برای بسیاری از انواع سیستم‌های تجاری ضروری نخواهد بود. هر پروژه باید با سطح

تست که واقعا به آن نیاز دارند و همچنین سطح و نوع مستندات آزمایشی که با تحمل ریسک و منابع تست موجود آن‌ها متناسب است، کنار بیاید.

۳-۹-۵- آزمایش و تایید الزامات

یک ویژگی مهم یک نیاز خوب این است که باید آزمایش پذیر باشد (که قابل تایید نیز نامیده می‌شود). آزمایش پذیری (یا تایید پذیری) این سوال را می‌پرسد: "آیا راهی وجود دارد که بتوان شرایط مورد نیاز را تأیید کرد؟ " اگر پاسخ "نه" باشد ، شرط تعیین شده فاقد اعتبار است و باید مجدداً آزمایش و بازنویسی شود.

زمانی که هر الزام نرم‌افزاری تعریف می‌شود، تیم تست باید روش‌های تایید مورد استفاده برای تایید اینکه چگونه هر الزام برآورده می‌شود را مشخص کند. چهار روش تایید معمول شامل بازرسی، تحلیل، نمایش و تست است. روش‌های تایید، و سطوح تایید، معمولا در SRS وIRS آمده است.

جدول ۳-۲ تست چک کردن تجزیه و تحلیل خطر نرم افزار

چه جنبه هایی از برنامه:	• برای مشتری مهمترین هستند؟ • آیا می توان در اوایل چرخه آزمایش، آزمایش کرد؟ • پیچیده ترند و به احتمال زیاد مسئول بیشتر خطاها هستند؟ • آیا در حالت عجله و وحشت توسعه یافته اند؟ • در پروژه مشابه / مرتبط قبلی مشکلاتی ایجاد شده است؟ • آیا هزینه های زیادی برای نگهداری در پروژه های مشابه قبلی داشته اید؟ • حاوی الزامات یا طراحی نامشخص یا ضعیف اندیشیده شده هستید؟
کدام قابلیت های عملکردی:	• آیا مهمترین هدف عملیاتی مورد نظر پروژه است؟ • بیشترین تأثیر ایمنی را دارید؟ • بیشترین تأثیر مالی را بر روی کاربران دارند؟
کدام نوع خاصی از آزمایشات:	• برنامه هایی که به عنوان جنبه های دارای بالاترین ریسک شناخته می شوند ، آدرس داده می شوند؟ • بیشترین شکایت از خدمات مشتری ایجاد می شود؟ • چندین ویژگی را پوشش می دهد؟ • آیا بهترین پوشش پرخطر را دارید؟

۳-۹-۶ – عملیات test-like-you-fly / test-like-toy

اگر سیستم نرم‌افزار محور شما کاملا با نیازهای مشتری شما سازگار باشد، و همه تست‌های صلاحیت را پشت سر بگذارد، اگر آن را در شرایط واقعی عملیاتی (پرواز مانند)و محیط مورد انتظار تست نکنید، باز هم ممکن است شکست بخورد.

عملیات test-like-you، که "test-like-you-fly" نیز نامیده می‌شود، روشی برای تست سیستم‌های نرم‌افزاری پیچیده است. این روش برای تست کردن، جایگزینی برای هر گونه کارکرد آزمون توصیف‌شده در این کتاب راهنما نیست. این یک روش آزمایشی اضافی و اختیاری است که در آن همه عناصر یا اجزای سیستم که با یکدیگر در ارتباط هستند، باید قبل از اینکه سیستم به کار گرفته شود، با هم تست شوند. ۴-۱۰-۹ برخی مثال‌های جالب از کل خطاهای سیستم، یا تجزیه در عملکرد سیستم را توصیف می‌کند که اگر رویکرد عملگر test-like-you مورد استفاده قرار گرفت، از آن اجتناب می‌شد. برخی از این نمونه‌های شکست به نظر من خلاصه‌ای از فقدان عقل سلیم ساده هستند.

۳-۹-۷- ارزش تست خودکار

رویکردهای عمومی برای تست اتوماسیون، مانند تست کد محور یا تست رابط کاربر گرافیکی، در اینجا پوشش داده نمی‌شوند اما شما باید با این رویکردها آشنا باشید. به عنوان SPM، شما باید به طور مستقیم با تصمیمات مربوط به این که چه چیزی را خودکار کنید، چه زمانی خودکارسازی کنید، یا حتی آیا پروژه شما واقعا باید خودکارسازی کند چون این تصمیمات مهم هستند. انتخاب ویژگی‌های درست محصول شما به عنوان کاندید اتوماسیون اغلب موفقیت ابتکار اتوماسیون را تعیین می‌کند.

برای پروژه‌های بزرگ، یا پروژه‌های طولانی‌مدت، تست خودکار می‌تواند بسیار مقرون‌به‌صرفه باشد. اما برای پروژه‌های کوچک‌تر، زمان مورد نیاز برای یادگیری و اجرای ابزارهای تست خودکار معمولا ارزشمند نیست. ابزارهای تست خودکار ممکن است تست کردن را آسان‌تر نکنند. یکی از مشکلات ابزارهای تست خودکار این است که اگر تغییرات مداومی در محصول یا سیستم شما در حال تست شدن وجود داشته باشد، لازم است که دستورالعمل‌ها اغلب به روز شوند، که این کار بسیار وقت گیر است. مشکل دیگر چنین ابزارهایی این است که تفسیر نتایج (صفحات نمایش، داده‌ها، log ها و غیره)نیز می‌تواند کاری زمان بر باشد. این را به عنوان رای مخالف تست خودکار تفسیر نکنید. فقط دقت کنید که این کار برای پروژه شما مقرون‌به‌صرفه باشد.

۳-۹-۸- زمان توقف تست

متغیرهای بسیاری در تصمیم‌گیری درباره میزان نیاز به تست برای تولید یک محصول نرم‌افزاری قابل‌اعتماد که کاملا به نیازهای مشتری شما پاسخگو باشد، وجود دارند. متغیرهای کلیدی در این فرآیند تصمیم‌گیری عبارتند از اندازه و پیچیدگی کد، توانایی تسترهای شما، به علاوه اهمیت و قابلیت اعتماد سیستم مورد نیاز. دلایل آشکار برای توقف آزمایش عبارتند از:

* وقتی تمام موارد آزمایش (یا درصد تایید شده مشتری) با موفقیت تکمیل می‌شوند

* هنگامی که مهلت زمانی انجام تست پروژه به دست می‌آید

*زمانی که بودجه برای تست کردن به طور کامل مصرف می‌شود

*وقتی که تشخیص نرخ تشخیص اشکال زیر سطح پذیرش از پیش تعیین‌شده قرار می‌گیرد.

* هنگامی که تست عملکرد به سطح مطلوبی از عملکرد تایید شده دست می‌یابد.

علاوه بر این، تکنیک‌های دیگری نیز وجود دارند که می‌توانند برای تخمین زمانی که تیم تست سطح قابل قبولی از تشخیص نقص را انجام داده‌است، مورد استفاده قرار گیرند. سه تکنیک نمونه در زیر توضیح داده شده‌اند: استفاده از اندازه‌گیری‌ها، ضبط - بازصید، و بارورسازی خطا.

تعریف پی‌گیری با استفاده از اندازه‌گیری. اندازه‌گیری و ردیابی عیوب می‌تواند برای تعیین زمان متوقف کردن تست مورد استفاده قرار گیرد. به عنوان مثال، SPM و مشتری ممکن است قبل از شروع آزمایش توافق کنند که هنگامی که نرخ تشخیص نقص زیر یک سطح مشخص پایین می‌آید، تلاش آزمایش کامل در نظر گرفته خواهد شد. برخی از اندازه‌گیری‌ها که می‌توانند برای ردیابی مداوم نقص مورد استفاده قرار گیرند عبارتند از:

* تعریف چگالی (تعداد عیوب در هر هزار خط کد)

*تعریف نرخ کشف (تعداد نقص‌های جدید و تعداد نقص‌های یافت‌شده در فازه‌ای درج نقص)

*تعریف نرخ تفکیک (تعداد نقص‌های ثابت شده)را تعریف کنید.

* تعداد موارد تست توسعه‌یافته، اجرا خشک، اجرا شده و پذیرفته

پی‌گیری تعداد کل خطاهای شناسایی‌شده کمک زیادی به شما نمی‌کند چون نمی‌دانید چه تعداد خطای کلی وجود دارد، پس نمی‌توانید درصد موفقیت خود را محاسبه کنید.

صید و بازصید. یک روش جالب برای تخمین تعداد عیوب در یک محصول نرم‌افزاری توسط هامفری (۲۰۰۰)پیشنهاد شد. او به دنبال یک روش ساده برای پیش‌بینی نواقص باقی مانده براساس نتایج دو یا چند منتقد مستقل بود که یا بازنگری‌ها، بازرسی‌ها و یا آزمایش‌ها را انجام می‌دادند. او دریافت که چنین روشی برای تخمین جمعیت حیواناتی که صید - بازصید نامیده می‌شوند، وجود دارد. یک نمونه از کاربرد این روش محاسبه تعداد ماهی‌های موجود در یک استخر است. برای مثال، شما به طور تصادفی ۳۰ ماهی بگیرید، آن‌ها را برچسب بزنید، و سپس آن‌ها را رها کنید. چند روز بعد ۲۵ ماهی به طور تصادفی از استخر صید می‌کنید و می‌بینید که پنج عدد از آن‌ها برچسب شما را دارند. معادله ساده ریاضی به صورت زیر است:

$$\text{150 ماهی درگیرشده در استخر} \approx \frac{\text{5 ماهی برچسب گذاری شده در نمونه}}{\text{25 ماهی درگیرشده در نمونه}} \approx \frac{\text{30 ماهی برچسب گذاری شده در استخر}}{\text{X (تعداد کل ماهی های استخر)}}$$

این روش می‌تواند برای تخمین عیوب در یک محصول نرم‌افزاری به کار رود که در آن مخزن به محصول نرم‌افزاری مورد آزمایش تبدیل می‌شود و ماهی به نقص تبدیل می‌شود. به عنوان مثال، یک مهندس نرم‌افزار یک محصول نرم‌افزاری را بازرسی می‌کند و عیوب پیدا شده را شناسایی و ثبت می‌کند. یک مهندس نرم‌افزار دوم همان محصول را بررسی می‌کند و برخی از نقص‌های تگ گذاری شده و نیز نقص‌های دیگر را پیدا می‌کند. با استفاده از همان فرمول بالا برای تعداد ماهی‌های موجود در استخر می‌توانید تعداد عیوب باقی مانده در نرم‌افزار را تخمین بزنید.

وقتی تعداد نقص‌های تخمین زده‌شده را با توجه به محاسبات پیدا می‌کنید، می‌توانید آزمایش را متوقف کنید. نتایج خوبی با استفاده از این روش گزارش شده‌است اما تنها در صورتی که بازنگری‌های خوبی داشته باشید و تعداد عیوب خیلی کم نباشد. (Davis, 2005)

خطا در مراقبت. یک روش جالب دیگر برای تعیین زمان توقف آزمایش، ایجاد برخی خطاها در کد (بدون گفتن به آزمایش کنندگان) است. هنگام استفاده از این روش بارورسازی خطا، مسیر نسبت خطاهای دانه‌دار که توسط تعداد کل خطاهای کاشته‌شده تقسیم شده‌اند را دنبال کنید. از این نسبت می توان به عنوان معیار پیشرفت آزمون استفاده کرد. وقتی همه خطاهای دانه‌دار پیدا شوند، شما می‌توانید تست را متوقف کنید چون از نظر آماری، تیم تست نشان داد که آن‌ها کار خوبی برای پیدا کردن نقص انجام داده‌اند. این می‌تواند یک رویکرد منطقی باشد تا زمانی که خطاهای دانه‌دار نسبتا دشوار باشند.