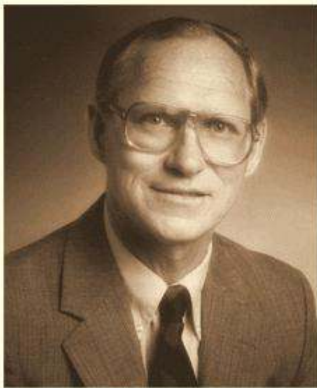
خلاصه‌ی تاریخچه‌ی مهندسی نرم‌افزار

نعیم اصفهانی ۸۴۲۰۱۰۰۳
esfahani A@T ce.sharif.edu

مقدمه

همان‌طوری که بزرگان ما گفته‌اند خواندن و فهمیدن تاریخ به ما درس‌هایی می‌دهد که می‌تواند بسیار آموزنده باشد. با کسب دانش از کارهایی که در گذشته انجام شده است می‌توان موفقیت‌های آن‌ها را تکرار کرد و از مواجهه با مشکلات آن‌ها خودداری نمود. در این نوشتار از زبان آقای بوهم[1] که یکی از تاثیرگذارترین افراد در شاخه‌ی مهندسی نرم‌افزار است تاریخچه‌ی آن را بررسی می‌کنیم.



افزایش دانش بشر به این ترتیب است که ابتدا چیزی را فرض می‌کند، سپس طی مدتی به این نتیجه می‌رسد که فرضی که انجام داده بود غلط است چون در برخی موارد اساسی اشتباه بوده است. در نهایت نیز از ترکیب فرض و متضاد آن به یک روش مخلوطی از آن‌دو می‌رسیم که مشکلات دو روش پایه‌ای را ندارد و در عوض خوبی‌های هردو را به ارث برده است. در این نوشتار این وضعیت به خوبی مشاهده می‌شود.

[1] Bohem

۱. دهه‌ی پنجاه

در این دوره همه‌ی افراد درگیر تولید در نرم‌افزار، مهندس سخت‌افزار یا ریاضی‌دان بودند. سخت‌افزار در صورت تولید و وجود مشکل به سختی قابل اصلاح است و معمولا لازم است که دوباره به طراحی و تولید بپردازیم. بنابراین در سخت‌افزار در مراحل قبل از تولید وسواسی‌تر عمل می‌شود: ″دوبار اندازه بگیر، یک بار ببر″. خواه نا خواه این تفکر بر روی نرم‌افزارها هم تاثیر گذاشت چون تولید کنندگان آن مهندسان سخت‌افزار بودند.
در تایید تفوق سخت‌افزار همین بس که نام دو جامعه‌ی تحقیقاتی کامپیوتر کاملا تحت تاثیر دید سطح پایین از کامپیوتر بود: انجمن ماشین‌های محاسب[۱]، جامعه‌ی کامپیوتر آی تریپل ای[۲].

۲. دهه‌ی شصت

هرچه در این دهه پیش می‌رفتیم شاهد این بودیم که نرم‌افزار خود را بیش‌تر به عنوان یک پدیده‌ی متفاوت نشان می‌دهد. حداق سه تفاوت اساسی بین نرم‌افزار و سخت‌افزار دیده شد:

- نرم‌افزار بر خلاف سخت‌افزار به راحتی پس از تولید تغییر پذیر است؛ به همین دلیل کم‌کم روش ″کد بنویس و اصلاح کن″[۳] باب شد.
- روش‌های سنجیدن تکیه‌پذیری[۴] مورد استفاده در سخت‌افزار تخمین خوبی در مورد نرم‌افزار ارائه نمی‌دادند.
- گسترش شدید نیاز به تولید نرم‌افزار باعث شد که افراد متخصص به اندازه‌ی کافی در دسترس نباشد.

تولید کنندگان نرم‌افزار که معمولا افرادی باهوش بودند کم‌کم به صورت افرادی در آمدند که از رنگ تعلق به سازمانی یا موسسه‌ای آزاد بودند. کار خود را می‌کردند و به گاوچران[۵] بیش‌تر شبیه بودند. این فرهنگ را زخمه‌ای[۶] می‌نامیدند و برنامه نویس‌ها به گرگ‌های تنها[۷] شبیه بودند. البته همه‌ی برنامه نویس‌ها این چنین نبودند و افرادی بودند که فریفته‌ی این فرهنگ نشدند.
تفاوت‌های دیگر این عصر عبارتند از:

- زیرساخت‌های بهتر، از کامپیوتر گرفته تا زبان برنامه نویسی
- برنامه‌های کوچک قابل مدیریت
- تاسیس دانشکده‌های علوم کامپیوتر و انفورماتیک

---

[1] Association for Computing Machinery (ACM)
[2] IEEE Computer Society
[3] Code and fix
[4] Reliability
[5] Cowboy Programmer
[6] Hacking Culture
[7] Lone Wolf

- شروع تولید نرم‌افزار برای ارزش افزوده
- افزایش برنامه‌های بزرگ ماموریت مبنا
- فاصله‌ی زیاد بین نیاز این نوع سیستم‌ها و توانایی‌های موجود

۳. دهه‌ی هفتاد

واکنشی که به روش "کد بنویس و اصلاح کن" نشان داده شد ظهور فرآیندی بود که در قالب آن کد نویسی سازماندهی شده بود و قبل از آن طراحی و قبل از طراحی، تحلیل گنجانده شده بود. در دو جهت شاهد این واکنش بودیم. از یک طرف روش‌های صوری[1] ظهور پیدا کردند که سعی بر اثبات درستی نرم‌افزار داشتند و از طرف دیگر شاهد ایجاد روش‌های ساخت یافته[2] و تولید از بالا به پایین بودیم.

پس از موفقیت نسبی برنامه نویسی ساخت یافته مفاهیم مشابهی نیز قوت گرفتند: ماژولار بودن[3]، چفت شدگی[4]، چسبندگی[5]، مخفی نمودن اطلاعات[6] و ساختار داده‌های مجرد[7] از جمله عمده‌ی این مفاهیم هستند.

در این میان رویس[8] روش آبشاری خود را مطرح کرد که در آن قبل از حرکت به فاز بعدی باید از اعتبار و صحت محصولات فازی که ترک می‌کردیم مطمئن می‌شدیم. در این روش، پروتوتایپ سازی قبل از ساختن کل سیستم مطرح بود اما به دلیل عدم فهم درست، مدل آبشاری کاملا ترتیبی ارائه شد و مشکلاتی را ایجاد کرد.

در نتیجه‌ی ارائه‌ی فرآیند قوی‌تر، روش‌های کمی قوی‌تری برای سنجیدن کارایی مهندسی نرم‌افزار مطرح شدند. کارهای متنوعی در این دهه انجام شد: بررسی جنبه‌های انسانی، بررسی برنامه‌ریزی تولید نرم‌افزار، ایجاد زبان‌های پاسکال و ماژولا-۲، تکنیک‌های مرور، خطوط تولید قابل استفاده‌ی مجدد و تکامل نرم‌افزاری از مباحث عمده‌ی مطرح شده در این عصر هستند.

در انتهای این دهه کم‌کم از محبوبیت مدل آبشاری و روش‌های صوری کاسته شد. عدم مقیاس پذیری روش‌های صوری و هزینه‌ی بالای مدل آبشاری (هزینه‌ی تولید نرم‌افزار که قبلا از تولید سخت‌افزار کم‌تر بود هم‌اکنون بیش‌تر از آن شده بود) باعث شد که این روش‌ها دور زده شوند و مدیران اصرار کنند که در تحلیل و طراحی وقت کم‌تری گذاشته شود تا زودتر به کد برسیم: "بهتره بجنبیم و کد زدن رو شروع کنیم تا وقت کافی برای دیباگ کردن را داشته باشیم!"

---

[1] Formal Method
[2] Structured Programming
[3] Modularity
[4] Coupling
[5] Cohesion
[6] Information Hiding
[7] Abstract Data Types
[8] Royce

۴. دهه‌ی هشتاد

مشکل عدم استفاده از فرآیند معمولا با ضمیمه کردن استانداردهایی به قراردادها تامین می‌شد. اما پس از مدتی وزارت دفاع آمریکا[1] به موسسه‌ی مهندسی نرم‌افزار[2] دانشگاه کارنگی ملون[3] کمک کرد تا مدلی برای نشان دادن بلوغ نرم‌افزار و توانایی آن[4] ایجاد کند تا بتوان با استفاده از آن فرآیند موجود در سازمان‌ها را ارزیابی کرد. از آن پس سازمان‌ها از ترس این‌که دیگر با آن‌ها قرار داد بسته نشود بر روی این استاندارد و استانداردهای مشابه سرمایه گذاری کردند.

در این میان ابزارهای تولید نرم‌افزار نیز محبوبیت خاصی پیدا کردن و سعی بر این بود که محیط یکپارچه‌ای از ابزارهای برای تولید نرم‌افزار ایجاد شود. این تلاش‌ها در نهایت به ابزارهای کامپیوتری کمکی مهندسی نرم‌افزار[5] منتهی شد.

بهبود فرآیند تولید نرم‌افزار تاثیر زیادی بر کاهش کارهای تکراری داشت اما در این فاز به دنبال کار بهینه‌تری بودند: عدم انجام اکثر کارها با استفاده از تولید خودکار کد از توصیف. اما با ارائه‌ی مقاله‌ای از بروک[6] مشخص شد که برخی کارها به هر حال باید توسط انسان انجام شوند و نمی‌توان همه‌چیز را اتوماتیک کرد.

بنا بر تحقیقی انتقال مفاهیم مهندسی نرم‌افزار به عمل در این دوره تاخیری ۱۸ ساله داشت؛ بنابراین وزارت دفاع این بار هم با استفاده از موسسه‌ی مهندسی نرم‌افزار سعی بر کاهش این زمان و سرعت بخشیدن به تغییرات تکنولوژیکی داشت.

در این دهه تکنولوژی به حدی رسید که واسط گرافیکی (طراحی به صورت "چیزی که می‌بینی همانی است که به دست می‌آوری") و بحث شبکه وارد کامپیوترهای شخصی شد.

استفاده‌ی مجدد در این دهه جلوه‌های متعددی دارد. حتی زیر ساخت‌هایی مانند چارچوب‌ها، سیستم عامل و ... که جلوی انجام کارهای تکراری سطح پایین را می‌گیرند جلوه‌هایی از آن هستند. چارچوب‌های استفاده‌ی مجدد و زبان‌های مخصوص دامنه‌ی نسل چهارم امکان استفاده‌ی مجدد را افزایش دادند. روش‌های شیءگرا حتی این استفاده‌ی مجدد را قوی‌تر نیز کردند. زبان‌های شیء گرای مطرح شده به گسترش شیءگرایی کمک کردند. در نهایت نیز طراحی و تولید شیءگرا با ظهور UML همگرا شد.

[1] Department of Defense (DoD)
[2] Software Engineering Institute (SEI)
[3] Carnegie Mellon University (CMU)
[4] SoftWare Capability Maturity Model (SW-CMM)
[5] Computer-Aided Software Engineering (CASE)
[6] Brook

۵. دهه‌ی نَوَد

در این دهه با مطرح شدن مفاهیمی مانند الگوهای طراحی، معماری نرم‌افزار و زبان‌های توصیف معماری پایه‌های شیءگرایی قوی‌تر شد. اهمیت رقابت در بازار نرم‌افزار با حضور اینترنت باعث شد که کم شدن زمان ارسال محصول به بازار اهمیت دو چندانی بگیرد. برای کاهش این زمان از روش‌های هم‌زمان مهندسی نرم‌افزار[۱] به جای روش‌های ترتیبی استفاده می‌شد. استفاده‌ی مجدد هم برای کمک به کاهش زمان ارسال به بازار مورد توجه قرار گرفت.

محصولاتی که با کاربر تعامل داشتند[۲] مفهوم دیگری که در این عصر به آن پرداخته شد؛ این محصولات را نمی‌توان توصیف کرد و در عوض باید خلق شوند. به طور مثال نیازمندی‌های مربوط به واسط گرافیکی کاربر[۳] تنها وقتی دیده می‌شوند قابل بیان هستند.

مدل مارپیچی برای تولید نرم‌افزار برای تولید هم‌زمان نرم‌افزار ارائه شد. برای جمع و جور کردن مجموعه کارهای موازی، نقاطی از زمان را به عنوان مراحلی از پروژه در نظر می‌گرفتند تا با شروط بله/خیر تطابق اجزایی که به صورت موازی در حال تولید هستند (نیازمندی، پروتوتایپ، معماری، برنامه و ...) را مشخص کنند.

تولید نرم‌افزارهای کد باز[۴] در این دهه با ایجاد سازمان نرم‌افزار مجانی توسط استالمن[۵] و استاندارد عمومی GNU مطرح شد. این جنبش که پایه‌های آن از فرهنگ زخمه‌ای دهه‌ی ۶۰ بود در سال ۱۹۸۵ رسمیت گرفت. نمونه محصول‌های موفق این جنبش عبارتند از: Linux، Apache، TCL، Python، Perl و Mozilla.

مفهوم دیگری که در این عصر مورد توجه قرار گرفت امکان ارائه‌ی نرم‌افزار برای افراد غیر برنامه‌نویس است. تحقیقاتی اساسی بر روی ارتباط انسان با ماشین[۶] صورت گرفت. در نهایت نیز محوریت تحقیق از چگونگی تعامل یک فرد با کامپیوتر به پشتیبانی از مجموعه‌ای از افراد و ارتباط آن‌ها با محصول انتقال یافت.

۶. دهه‌ی صفر

در دهه‌ی پایانی هزاره چیزی که مد نظر است سرعت تولید نرم‌افزار است. سرعت تغییر تکنولوژی باعث می‌شود که روش‌های کند (برنامه ریزی سنگین، با مستند سازی و توصیف زیاد سیستم) کارایی خود را از دست بدهند. در این میان روش تولید سریع برنامه[۷] مورد توجه قرار گرفت.

[1] Concurrent Software Engineering
[2] User Interactive Products
[3] Graphical User Interface (GUI)
[4] Open Source
[5] Stallman
[6] Human Computer Interaction (HCI)
[7] Rapid Application Development (RAD)

در اواخر دهه‌ی پیشین تعدادی متدولوژی چابک پا به عرصه‌ی وجود گذاشتند. این متدولوژی‌ها سرعت تولید سیستم را کاهش می‌دادند. پس از مدتی معلوم شد که این نوع از متدولوژی‌ها تنها در پروژه‌های کوچک خوب عمل می‌کنند و وقتی صحبت از پروژه‌های بزرگ می‌شود این نوع از متدولوژی‌ها خیلی کارا نیستند.

سازمان‌ها کم‌کم به این سمت حرکت کردند که به جای تطابق افراد با تکنولوژی، تکنولوژی را با افراد منطبق کنند. استفاده‌پذیری محصول و ارزش افزوده‌ی آن جلوه‌ی بیش‌تری از تعداد خصوصیات و قیمت محصول که قبلا مهم‌تر بود پیدا کرد. روش‌هایی که ارزش را مبنا قرار می‌دهند سعی می‌کنند چارچوبی ارائه دهند که به وسیله‌ی آن بتوان تعیین کرد که در کجای پروژه از روش‌های چابک استفاده کرد و در کجا روش‌های سنگین‌تر را به کار برد. این امر به وسیله‌ی میزان ریسک و پویا بودن آن قسمتی از پروژه مشخص می‌شود.

به وجود آمدن نیازمندی در وسط پروژه با استفاده از روش‌های مورد استفاده در گذشته (آبشاری ،رسمی و حتی مدل بلوغ نرم‌افزار) هم‌خوانی ندارد. این وضعیت در پروژه‌های این دهه به وجود آمده است؛ بنابراین نیاز به متدولوژی‌های انعطاف‌پذیرتر و بر مبنای ریسک بیش‌تر شده است.

تکیه‌ی سازمان‌ها بر نرم‌افزار بیش‌تر شده است. با وجود اهمیت تکیه‌پذیری نرم‌افزار، این موضوع هنوز به اولویت اول تحقیقات مبدل نشده است. پیش‌بینی می‌شود که تا سال ۲۰۲۵ فاجعه‌ای به بزرگی یازده سپتامبر به دلیل بی توجهی به این امر اتفاق بیافتد - البته در کشورهایی که نرم‌افزار نقش حیاتی دارد - احتمالا پس از چنین اتفاقی این بحث به اولویت اول تحقیقات تبدیل شود.

تعداد اجزا و سیستم‌های تجاری موجود[1] در دسترس بیش‌تر شده است. این امر دو روی خوب و بد دارد. طرف خوب سکه، سرعت بالای تولید سیستم و تشابه منحنی رشد آن به خوبی سخت‌افزار است. منحنی رشد تولید نرم‌افزار خیلی نزدیک به تعداد مدارهای مجتمع و یا تعداد بسته‌ها[2] در اینترنت است. هزینه‌ی پرداختی برای هر خط کد ماشین در نرم‌افزار به این وسیله کاهش داشته است. اما طرف بد سکه، عدم دسترسی به کد جزئی از سیستم باعث می‌شود وابستگی پذیری سیستم تحت تاثیر آن مولفه باشد و نیز الگوی تکامل آن قسمت که توسط سازنده‌ی آن دیکته می‌شود بر ریسک برنامه‌ریزی تکامل نرم‌افزار می‌افزاید.

خیلی از وقت تولید کنندگان نرم‌افزار مصرف تنظیم و کنار هم قرار دادن این اجزای تجاری موجود می‌شود. از آن‌جا که در بازار این اجزا رقابت وجود دارد، هیچ دو محصولی مشابه هم یافت نمی‌شود. در ضمن هر ۱۰ ماه نسخه‌ی جدیدی از این محصولات ارائه می‌شود و پس از سه نسخه پشتیبانی از محصول قبلی متوقف می‌شود. به این دلیل استفاده و عیب‌یابی این محصولات کمی سخت است. در مقابل اجزای کد باز راحت‌تر عیب‌یابی می‌شوند و پشتیبانی آن‌ها نیز معمولا متوقف نمی‌شود.

پیدایش معماری سازمانی[3] و پیاده‌سازی مبتنی بر مدل[4] امکان تطابق بین اجزای تجاری موجود را بیش‌تر کرده است. به وسیله‌ی این روش مدل دامنه‌ای (بانک، خودرو، زنجیره‌ی مصرف و ...) را می‌سازیم. این مدل منجر

[1] Commercial-of-the-shelf (COTS)
[2] Packet
[3] Enterprise Architecture
[4] Model Driven Development (MDD)

به یک معماری می‌شود که اجزای آن دارای چسبندگی[1] بالا و جفت‌شدگی[2] پایین هستند. به این ترتیب امکان تولید سریع سیستم‌های تکیه‌پذیر در یک دامنه را خواهیم داشت. تنها مشکلی که این روش‌ها دارند، تغییر سریع زیرساخت‌ها (مانند موبایل) و بازارایی ساختار دامنه‌هاست.

بسیاری از شکست‌های پروژه‌های نرم‌افزاری ریشه در مهندسی سیستم (۶۶ درصد از مشکلات از کمبود ورودی‌های کاربران، نیازمندی‌های ناقص و متغیر، انتظارات و زمان‌بندی‌های نامعقول، هدف‌های نامشخص و کمبود حمایت اجرایی است) دارد. به نظر می‌رسد مهندسی نرم‌افزار و مهندسی سیستم با هم در آینده‌ی نزدیک یکپارچه شوند.

۷. دهه‌ی ده

از هشت گرایش مطرح شده در مهندسی نرم‌افزار پنج مورد را تا کنون مورد بررسی قرار دادیم:

- تغییرات سریع و نیاز به متدولوژی‌های چابک‌تر
- تاکید بیش‌تر بر روی استفاده و ارزش نرم‌افزار
- مهم شدن نرم‌افزار و اهمیت تکیه‌پذیری آن
- افزایش استفاده از مولفه‌های تجاری موجود، استفاده‌ی مجدد و یکپارچگی با سیستم‌های موجود
- افزایش یکپارچگی مهندسی نرم‌افزار و مهندسی سیستم

دو عدد دیگر از این گرایش‌ها در این بخش (دهه‌ی ۲۰۱۰) مورد بررسی قرار می‌گیرند:

- ارتباط جهانی
- سیستم‌هایی از سیستم‌ها

مورد باقی‌مانده به همراه دو گرایش جدید نیز در بخش بعدی مطرح خواهند شد:

- افزایش توان محاسباتی
- بیو انفورماتیک[3]
- خود مختاری[4]

امکان پخش کار مستقل از محل پایه‌ای برای همکاری‌های با سینرژی بالاتر را ایجاد کرده است. روی دیگر سکه در این مورد چالش‌های مربوط به همگام شدن می‌باشد. بحث برون سپاری[5] به بحثی داغ تبدیل شده است و بنا بر کتاب معروف فریدمن[6] دنیا مسطح شده است؛ کاری که قرار است در یک جای دنیا ارائه شود در جای دیگر

[1] Cohesion
[2] Coupling
[3] Bio-Computing
[4] Autonomy
[5] Outsourcing
[6] Friedman

دنیا و در شیفت‌های کاری مکمل (سه شیفت کاری) انجام می‌شوند و در نتیجه سرعت تولید سیستم‌ها بالاتر رفته است. این امکان نیازهای جدیدی را به وجود آورده است که از آن جمله می‌توان به تطابق فرهنگ‌ها[1] و بومی‌سازی[2] اشاره کرد. عمده چالش‌هایی که در این مورد مطرح خواهند شد عبارتند از:

- برقراری پل بین فرهنگ‌ها
- ایجاد دید مشترک و برقراری اعتماد
- هماهنگی بین چند حوزه‌ی زمانی
- مسائل مربوط به کار تیمی حساس به فرهنگ
- نحوه‌ی تنظیم قرارداد

سیستم‌های این دهه از مجموعه‌ای از سیستم‌ها که شاید توسط شرکت‌ها و سازمان‌های متفاوتی تولید شود تشکیل خواهند شد. بنابراین جدا دیدن این سیستم‌ها که بعدا قرار است به هم وصل شوند باعث تاخیرهای غیر قابل قبول در ارائه‌ی سرویس و مشکلات برنامه‌ریزی می‌شوند. برای غلبه بر این مشکل مفهوم سیستمی از سیستم‌ها مطرح می‌شود تا امکان یکپارچه شدن تعدادی سیستم که به صورت مستقل ایجاد شده‌اند را بدهیم. این ساختار باید علاوه بر مساله‌ی اصلی ذکر شده امکان کار با سیستم‌های بزرگ، بلوغ پویا و ایجاد نیازهای جدید را نیز فراهم کند.

۸. دهه‌ی بیست

افزایش توان محاسباتی که با توجه به قانون مور[3] پیش‌بینی می‌شود از دو جهت باعث گسترش نرم‌افزارها می‌شود. از طرفی سکوهای محاسباتی جدید (مانند ذرات هوشمند) معرفی خواهند شد و از طرف دیگر نوع برنامه‌های جدید (مانند برنامه‌ی یک عضو مصنوعی) مطرح خواهند شد. نگهداری، تنظیم، درستی‌یابی و ... این نوع نرم‌افزارهای جدید به تحقیقاتی وسیع نیازمند است.

به نظر می‌رسد افزایش توان محاسباتی باعث ایجاد روش‌های مهندسی نرم‌افزار جدید و قوی‌تری شوند. ابزارهای مهندسی نرم‌افزار با کاربر ارتباط برقرار می‌کنند و اطلاعات او را افزایش می‌دهند و به او فیدبک تصمیماتی را که گرفته ارائه می‌کنند؛ این ابزارها نقشی مانند کمربند ایمنی یا کیسه‌ی هوا در ماشین را بازی خواهند کرد.

خودمختاری عبارت است از پیشرفت‌های تکنولوژیکی که با استفاده از افزایش توان محاسباتی کامپیوتر و نرم‌افزار را قادر می‌کنند که به صورت خود محور شرایط را بررسی کند و بهترین تصمیم را بگیرد.

[1] Culture Matching
[2] Localization
[3] Moore's Law

ارتباط بیولوژی و کامپیوتر می‌تواند از دو جنبه مطرح شود. در جنبه‌ی اول بیولوژی به ما کمک می‌کند تا مسائلی را که قبلا نمی‌توانستیم حل کنیم را حل کنیم. در جنبه‌ی دیگر کامپیوتر به بیولوژی کمک خواهد کرد؛ اعضای مصنوعی و یا بهبود کارکرد داروها از نمونه‌های این رابطه هستند.

نتیجه‌گیری

در جدول زیر لیست نتایجی که در هر دهه گرفته شده است را به طور خلاصه آورده‌ایم. برای هر دهه دو موردی که ضرورت آن مشخص شد و یک مورد که ناکارآمد بودن آن مشخص شد را آورده‌ایم. البته این عبارت‌ها نمادین هستند:

دهه	تجربه‌ی موفق اول	تجربه‌ی موفق دوم	تجربه‌ی ناموفق
از دهه‌ی پنجاه	علم را نادیده نیانگارید	قبل از پریدن به محل هدف نگاهی داشته باید	از فرآیندهای سفت و سخت ترتیبی پرهیز کنید
از دهه‌ی شصت	به خارج از محدوده‌ی موجود نیز فکر کنید	به تفاوت‌های نرم‌افزار احترام بگذارید	از برنامه نویسی گاوچران مابانه پرهیز کنید
از دهه‌ی هفتاد	خطاها را زودتر از بین ببرید	هدف سیستم را مشخص کنید	در استفاده از روش بالا به پایین زیاده‌روی نکنید
از دهه‌ی هشتاد	برای افزایش بهره‌وری راه‌های زیادی وجود دارد	چیزی که برای محصول خوب باشد برای فرآیند هم خوب خواهد بود	خیلی به راه‌حل عمومی مطمئن نباشید
از دهه‌ی نود	زمان پول است	نرم‌افزار باید برای استفاده‌ی آدم مناسب گردد	سریع باشید ولی عجله نکنید
از دهه‌ی صفر	اگر تغییرات زیاد هستند، انعطاف‌پذیری چاره است	نیازها و نظرات تمام ذینفعان مهم را باید فهمید	با شعارهای خود ارتباط عاطفی برقرار نکنید!
برای دهه‌ی ده	چیزهایی که دارید و می‌توانید را به هم متصل کنید	راه حل جایگزین داشته باشید	هر چیزی که می‌خوانید را باور نکنید

جدول۱. خلاصه‌ی نتایج بدست آمده در هر دهه

در مورد آموزش مهندسی نرم‌افزار هم باید موارد زیر را مد نظر قرار داد:

- مطالب دروس باید به روز نگهداری شوند
- روندها را پیش‌بینی کنیم و دانش‌جویان را برای مقابله با آن‌ها آماده کنیم
- قوانین موجود را بررسی کنیم و قدیمی‌ها را حذف و جدیدها را اضافه کنیم
- بسته‌های آموزشی با مقیاس کوچک برای کسب تجربه‌ی پروژه‌های بزرگ‌تر آماده کنیم
- در لبه‌ی تحقیق و تکنولوژی مهندسی نرم‌افزار حرکت کنیم و نتایج را در دروس انعکاس دهیم
- به دانشجویان چگونه یاد گرفتن را یاد دهیم
- یادگیری عمری را ارائه دهیم، مهندسان نرم‌افزار کار خود را تمرین خواهند کرد