

It Ain't Broke: למה יסודות הנדסת התוכנה חשובים עכשיו יותר מתמיד

Matt Pocock טוען ש-specs-to-code מייצר זבל, שקוד גרוע הוא היקר ביותר שהיה אי פעם, ומציג חמישה skills ל-Claude Code שמבוססים על ספרי הנדסת תוכנה קלאסיים

· Matt Pocock · AI Engineer Europe (לפי הגרפיקה בשקפים; הסרטון הועלה מחדש ב-X על ידי Mark Ajzenstadt) · הועלה ב-X ב-2026-07-09, תאריך ההרצאה המקורית לא ידוע

משך: כ-18 דקות · X

סיכום אוטומטי — AI Briefing (aibriefing.dev)

על המסמך הזה

זו הרצאה של כ-18 דקות של Matt Pocock, שמציג את עצמו כמורה ומלמד קורס בשם Claude Code for Real Engineers. לפי הכיתוב התחתון בשקפים, ההרצאה ניתנה ב-Google AI Engineer Europe (בחסות Google DeepMind), ושמה It Ain't Broke: Why Software Fundamentals Matter More Than Ever. הסרטון שממנו נכתב המסמך הוא העלאה חוזרת ב-X, ולכן הכותרת והתיאור של מי שהעלה אותו לא שימשו כמקור.

הטיעון המרכזי: תנועת specs-to-code, שבה כותבים מפרט, נותנים ל-AI לייצר קוד ולא מסתכלים על הקוד עצמו, מובילה לשחיקה מתמשכת של בסיס הקוד. לכן "code is cheap" הוא רעיון שגוי. בסיס קוד שקשה לשנות מונע מכם ליהנות ממה ש-AI יכול לתת. מכאן Pocock עובר על סדרה של failure modes מוכרים בעבודה עם AI, ולכל אחד מצמיד פתרון מספר קלאסי (Ousterhout, The Pragmatic Programmer, Brooks, DDD, Kent Beck) ו-skill (שכתב בעצמו).

איך לקרוא את המסמך

- 1. **תקציר מנהלים:** הטענות המרכזיות עם חותמות זמן, ומה לא נאמר בהרצאה.
- 2. **התזה:** למה specs-to-code נכשל ולמה קוד גרוע יקר.
- 3. **יישור קו עם ה-AI: Grill Me** ו-ubiquitous language (1 failure modes ו-2).
- 4. **Feedback loops ו-TDD:** כשה-AI בנה את הדבר הנכון, אבל הוא לא עובד.
- 5. **Deep modules:** איך בסיס קוד בנוי כדי שיהיה קל לבדוק אותו ולנווט בו.
- 6. **העומס הקוגניטיבי והתפקיד האסטרטגי:** Design the interface, delegate the implementation.
- 7. **מה עושים עכשיו:** צעדים מעשיים וטבלת שאלות פתוחות.
- 8. **מילון מונחים.**

הערות מקור המסמך הופק אוטומטית מהתמליל ומהשקפים של ההקלטה. שמות ומונחים עלולים להשתבש בתמלול האוטומטי: למשל, בתמליל מופיע Osterhout, והשקף מראה שהשם הנכון הוא "Claw Code"; Ousterhout הוא Claude Code; בתחילת התמליל מופיע שם לא קשור שהוא כנראה ארטיפקט של התמלול. הציטוטים מועתקים מילה במילה ובבדקו מול חותמת הזמן שלהם. לא נוסף מידע ממקורות חיצוניים.

1. תקציר מנהלים

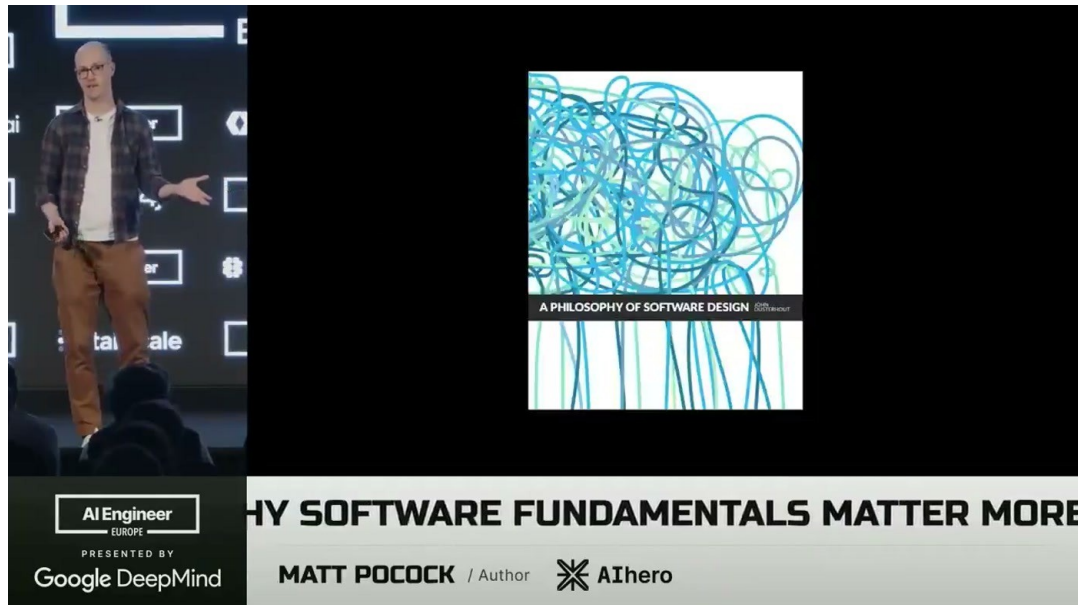
Pocock לא מציע כלי חדש. הוא טוען שהבעיות שאנחנו נתקלים בהן מול AI הן בעיות ותיקות של הנדסת תוכנה, ושהפתרונות כבר כתובים בספרים. ה-AI הוא מתכנת טקטי טוב, ואחריות הדיזיין נשארת אצל המהנדס.

- **specs-to-code מייצר קוד שהולך ומתדרדר:** הוא ניסה להריץ את ה"קומפילר" שוב ושוב בלי להסתכל על הקוד, וכל ריצה נתנה קוד גרוע יותר, עד שקיבל "garbage". הוא מכנה את הגישה vibe coding בשם אחר [1:34].
- **קוד גרוע יקר יותר מאי פעם:** בסיס קוד שקשה לשנות מונע ממך לנצל את ה-AI, ו-AI מצליח מאוד דווקא בבסיס קוד טוב [3:05].
- **Grill Me במקום plan mode:** skill בן כמה שורות שגורם ל-AI לשאול 40 עד 100 שאלות עד שנוצר design concept משותף. לפי Pocock, ל-repo יש בערך 13,000 כוכבים [4:39], [6:10].
- **שפה משותפת מקצרת את ה-skill:** AI שסורק את בסיס הקוד ומייצר קובץ markdown של ubiquitous language. לדבריו, זה משפר את התכנון, מקצר את ה-thinking ומקרר את המימוש לתוכנית [7:42], [9:13].
- **Feedback loops הם תנאי בסיס:** TypeScript, גישה לדפדפן לאפליקציות front-end ו-automated tests. ה-AI לא מנצל אותם היטב ונוטה לייצר הרבה קוד בבת אחת [9:13].
- **TDD כופה צעדים קטנים:** "TDD". The rate of feedback is your speed limit מכריח את ה-LLM לכתוב טסט, להעביר אותו ורק אז לעשות refactor [10:46].
- **Deep modules הופכים את הקוד לבדיק ולקריא ל-AI:** הרבה מודולים קטנים ורדודים קשים ל-AI לניווט. מודולים עמוקים עם ממשק פשוט מאפשרים לבדוק דרך הממשק [12:20], [13:51].
- **Design the interface, delegate the implementation:** מתייחסים למודולים לא קריטיים כ-grey boxes, ובכך חוסכים עומס קוגניטיבי. זה לא מתאים לתחומים כמו finance [15:22].
- **המהנדס הוא הדרג האסטרטגי:** "Invest in the design of the system every day" (Kent Beck). לפי Pocock, specs-to-code הוא divesting מהדיזיין [16:55].

מה לא נאמר

- **אין מדידות:** אף טענה על שיפור (פחות verbose, מימוש מדויק יותר, פחות עייפות) לא מגובה במספרים או בהשוואה. הכול מבוסס על ניסיון אישי.
- **13,000 כוכבים "or something":** המספר ניתן כהערכה. מעבר לזה לא הוצג נתון על אימוץ או על תוצאות.
- **skill הארכיטקטורה לא הוסבר לעומק:** הוא מודה שזה "quite complicated" ומתאר רק את הרעיון הכללי.
- **DDD בשלבי חקירה:** Pocock אומר במפורש שהוא "DDD" still kind of on the edge of exploring.
- **אין התייחסות לעלות:** לא נאמר דבר על טוקנים, זמן ריצה או עלות של 100 שאלות grilling.
- **המספור לא עקבי:** בהרצאה מופיעים failure mode number six ו-tip number five בלי שהוצגו כל המספרים שביניהם. כנראה דילוג בהרצאה או קיצור בעריכה.

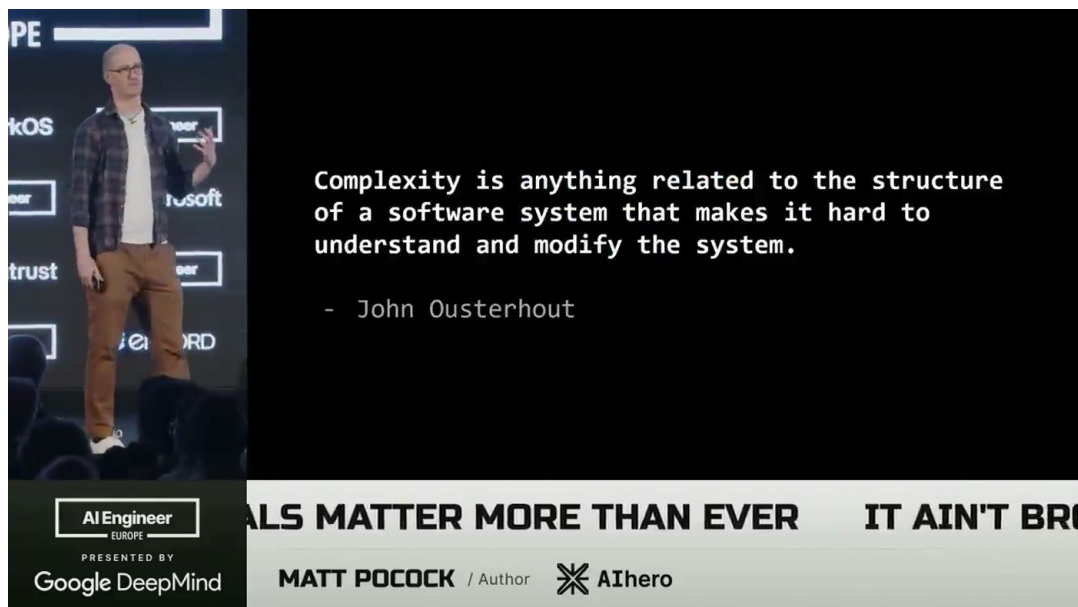
2. התזה: specs-to-code נכשל, וקוד אינו זול



עטיפת הספר A Philosophy of Software Design של John Ousterhout, עם איור של קווים מסובכים שמייצגים complexity

Pocock פותח בתנועת specs-to-code: כותבים מפרט, ה-AI הופך אותו לקוד, וכשיש בעיה מתקנים את המפרט ומריצים שוב, בלי לגעת בקוד. הוא ניסה את זה ומספר שכל איטרציה הובילה לקוד גרוע יותר: "I kept running the compiler, kept running the compiler, and I would just end up with garbage" [1:34]

כדי להסביר ל-LLM מהו קוד טוב, הוא חזר לשני ספרים. מ-Ousterhout לקח את ההגדרה של complexity: בסיס קוד רע הוא בסיס קוד שקשה לשנות. מ-The Pragmatic Programmer לקח את המושג software entropy: כל שינוי שנעשה בלי לחשוב על הדיזיין של המערכת כולה מדרדר את בסיס הקוד, וזה בדיוק מה שהריצות החוזרות של specs-to-code עשו.



ציטוט של John Ousterhout: complexity היא כל דבר במבנה המערכת שמקשה להבין אותה ולשנות אותה

Matt Pocock [3:05] I think code is not cheap. In fact, bad code is the most expensive it's ever been. Because if you have a code base that's hard to change, you're not able to take all of the bounty that AI can offer

3. יישור קו עם ה-AI: ubiquitous language-

1 Failure mode הוא מצב שבו ה-AI לא עשה את מה שרציתם. Pocock מתאר אותו כבעיה של requirements gathering. מ-The Design of Design של Frederick P. Brooks הוא לקח את המושג design concept: רעיון משותף ובלתי נראה של מה שבונים, "[4:39] not something you can put in a markdown file". האבחנה שלו היא שלו ול-AI אין design concept משותף.

הפתרון הוא ה-skill שנקרא Grill Me, וכולו כמה שורות: "Interview me relentlessly about every aspect of this" [4:39] plan until we reach a shared understanding. walk down each branch of the design tree" [4:39] ה-AI שואל 40, 60 ולפעמים 100 שאלות. את השיחה שנוצרת הופכים ל-PRD, או ישירות ל-issues כשמדובר בשינוי קטן, ומשם AFK agent ממשיך. Pocock מעדיף את זה על plan mode של Claude Code: "Plan mode is extremely eager to create an asset" [6:10].

2 Failure mode הוא AI מילולי מדי. Pocock מושה את זה לעבודה עם domain expert שמדבר בשפה שאינכם מכירים. מ-DDD הוא לקח את ה-ubiquitous language: קובץ markdown עם רשימת מונחים משותפים שמשתמשים בהם בקוד, בשיחות ובעבודה מול ה-AI. ה-skill שלו סורק את בסיס הקוד, מחלץ את הטרמינולוגיה ומייצר טבלאות markdown. הקובץ פתוח אצלו כל הזמן בזמן grilling. לפי מה שראה ב-thinking traces, זה גם משפר את התכנון וגם גורם ל-AI "to think in a less verbose way" [9:13].

4. TDD Feedback loops-

3 Failure mode: ה-AI בנה את הדבר הנכון, אבל הוא לא עובד. הצעד הראשון הוא לתת לו feedback loops: front-end automated tests-ו- static types ("If you're not using TypeScript, that's crazy" [9:13]) גישה לדפדפן כשבונים type check כך נזכר להריץ type check.

ב-The Pragmatic Programmer קוראים לזה outrunning your headlights. מכאן ה-skill השלישי, TDD, שכופה צעדים קטנים: טסט, מעבר, Pocock refactor. מודה שכתובת טסטים תמיד הייתה קשה, כי ההחלטות תלויות זו בזו: מה גודל ה-unit שבודקים, מה עושים לו mock ואילו התנהגויות בודקים. המסקנה שלו מחזירה את הדיון לאיכות הקוד.

— Matt Pocock [10:46] The rate of feedback is your speed limit, which means that you should be testing as you go, taking small deliberate steps.

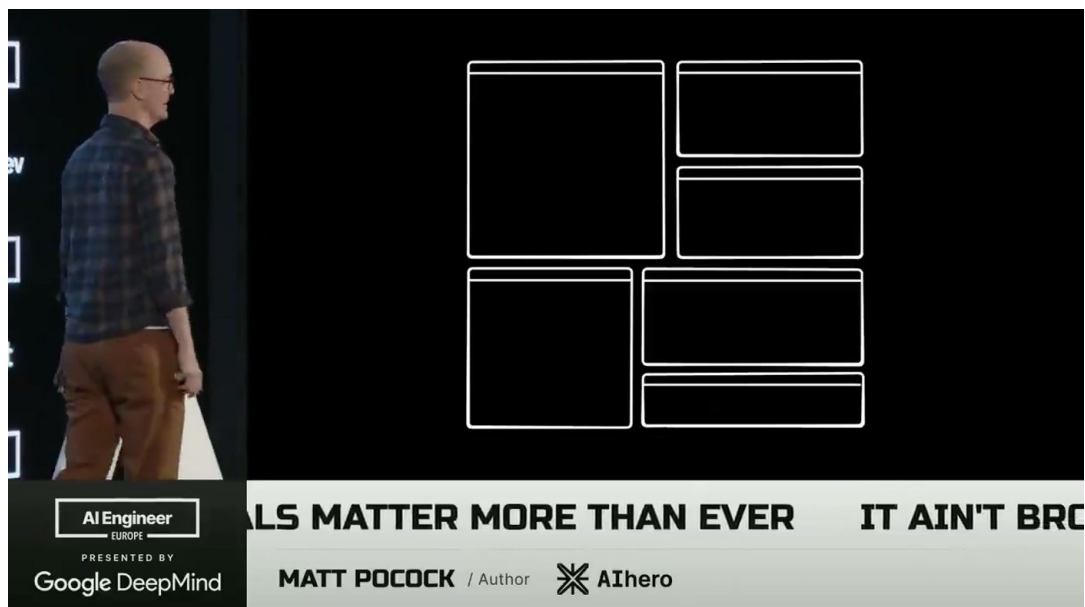
— Matt Pocock [10:46] what we notice is that good code bases are easy code bases to test.

5. Deep modules: בסיס קוד שה-AI מצליח לנווט בו

כדי להסביר איך נראה בסיס קוד בדיק, Pocock חוזר ל-relatively few large deep modules with-ו- shallow module-ב. [12:20] simple interfaces" [12:20] יש הרבה פונקציונליות מאחורי ממשק פשוט. ב-shallow module יש מעט פונקציונליות וממשק מסובך. לטענתו, AI נוטה לייצר בסיסי קוד מהסוג השני, ושם הוא גם מתקשה לחקור: הוא לא מגיע למודול הנכון בזמן ולא מבין את התלויות.



בסיס קוד של shallow modules: רשת צפופה של עשרות ריבועים קטנים שה- AI צריך לעבור ביניהם



אותו קוד מאורגן כ-deep modules: מספר קטן של בלוקים גדולים עם גבולות ברורים

במבנה של deep modules, את הממשקים "you should probably have a lot of control over", ואת המימוש "you can kind of leave that to the AI a bit" [13:51]. כדי לעבור מהמבנה הראשון לשני כתב Pocock skill במשך improve codebase architecture: חוקרים את בסיס הקוד, מאתרים קוד קשור ועוטפים אותו ב-deep module. התוצאה היא בסיס קוד "that rewards TDD", כי בודקים ומאמתים דרך הממשק.

6. העומס הקוגניטיבי והתפקיד האסטרטגי

הבעיה האחרונה (בהרצאה היא מכונה failure mode number six): אתם שולחים יותר קוד מאי פעם, והמוח לא עומד בקצב. "Raise your hand if you felt more tired than you have ever before in your development career" [13:51]. לטענת Pocock, בסיס קוד של shallow modules מכביד גם עליכם, ולא רק על ה- AI. ב-deep modules אפשר להתייחס למודולים כ-grey boxes: לתכנן את הממשק ולא לסקור את המימוש לעומק. הוא מסייג שזה מתאים לחלקים פחות קריטיים, ולא למשל ל-finance.

— **Matt Pocock [15:22]** So that's tip number five. Design the interface. Delegate the implementation

לכן מפת המודולים צריכה להיות חלק מה- ubiquitous language ומ-skills התכנון. ב-skill שכותב Pocock, PRD מפרט אילו מודולים משתנים ואיך הממשקים שלהם משתנים. הוא סוגר עם Kent Beck ועם מטאפורה צבאית: ה- AI

הוא "a sergeant on the ground", ומעליו צריך מישהו שחושב ברמה האסטרטגית [16:55]. ה-skills נמצאים ב-GitHub repo שהוא מכנה Matt Pocock skills, ואת הקורסים שלו אפשר למצוא ב-AIHero.dev.

Matt Pocock [16:55] Invest in the design of the system every day. And this is the core of it, right? — .Because specs to code, we are not investing in the design of the system. We are divesting from it

7. מה עושים עכשיו

- **לפני plan mode, להריץ grilling:** לבקש מה-agent לתחקר אתכם על כל ענף בעץ ההחלטות עד שיש הבנה משותפת, ורק אחר כך לייצר PRD או issues.
- **לייצר ubiquitous language:** קובץ markdown של מונחי הדומיין, שנוצר מסריקה של בסיס הקוד ומוזן לכל סשן תכנון.
- **להקשיח את ה-TypeScript feedback loops:** גישת דפדפן ל-agent בפרויקטי front-end, ו-TDD כדי לכפות צעדים קטנים.
- **למפות shallow modules:** לאתר אשכולות של קוד קשור ולעטוף אותם ב-deep modules עם ממשק פשוט שאפשר לבדוק.
- **לחלק אחריות:** אתם מתכננים ומבקרים את הממשקים, ה-agent כותב את המימוש. בקוד קריטי (למשל finance) ממשיכים לסקור גם את המימוש.
- **לכתוב את מפת המודולים בתוך ה-PRD:** בכל תכנון לפרט אילו מודולים וממשקים משתנים.

שאלה פתוחה	למה זה חשוב
האם grilling של 100 שאלות משתלם במשימות קטנות?	ההרצאה לא מתייחסת לזמן או לעלות. Pocock רק מציין שבשינוי קטן אפשר לעבור ישר ל-issues.
איך מודדים שה-ubiquitous language באמת מקצר את ה-AI?	הטענה נשענת על קריאת thinking traces, ולא הוצגה מדידה.
עד כמה skill הארכיטקטורה אמין בבסיס קוד גדול?	Pocock מודה שהתהליך מסובך, ולא הוצגו דוגמה או תוצאות.
איפה עובר הגבול בין grey box לקוד שחייבים לסקור?	הוא נתן רק דוגמה אחת (finance), בלי קריטריון כללי.
האם specs-to-code יכול לעבוד אם יש גם שכבת דיזיין?	Pocock פוסל את הגישה על סמך ניסיון אישי ולא בוחן גרסאות היברידיות.

8. מילון מונחים

מונח	משמעות
specs-to-code	גישה שבה מתחזקים רק את המפרט ונותנים ל-AI לייצר מחדש את הקוד, בלי לקרוא אותו
complexity	לפי Ousterhout: כל מה שבמבנה המערכת מקשה להבין אותה ולשנות אותה
software entropy	הנטייה של בסיס קוד להידרדר כשמשנים אותו בלי לחשוב על הדיזיין הכולל
design concept	לפי Brooks: הרעיון המשותף והבלתי נראה של מה שבונים, שאי אפשר לשמור כקובץ
design tree	עץ ההחלטות התלויות זו בזו, ש-Grill Me עובר עליו ענף אחרי ענף
Grill Me	skill שמורה ל-AI לתחקר את המשתמש עד שנוצרת הבנה משותפת
ubiquitous language	מונח מ-DDD: שפה אחידה למפתחים, לקוד ולמומחי הדומיין, ובהקשר הזה גם ל-AI
DDD	Domain-Driven Design, מתודולוגיה שבונה את הקוד סביב מודל הדומיין
PRD	Product Requirements Document, מסמך דרישות מוצר
AFK agent	agent שעובד לבד על issues בזמן שהמפתח לא ליד המחשב
plan mode	מצב תכנון ב-Claude Code שמייצר תוכנית לפני שמתחילים לבצע

מונח	משמעות
feedback loop	מנגנון שמחזיר ל-agent אות על נכונות הקוד: types, טסטים, דפדפן
outrunning your headlights	מ-The Pragmatic Programmer: להתקדם מהר יותר מקצב ה-feedback
TDD	Test-Driven Development: כותבים טסט, מעבירים אותו, ואז עושים refactor
deep / shallow module	מודול עמוק מסתיר הרבה פונקציונליות מאחורי ממשק פשוט, ומודול רדוד נותן מעט פונקציונליות עם ממשק מסובך
grey box	מודול שמתכננים ובודקים דרך הממשק שלו בלי לסקור את המימוש לעומק