Inżynieria oprogramowania

Metodyki adaptacyjne (giętkie / elastyczne / *agile*)

Wydział Matematyki, Informatyki i Mechaniki
Uniwersytet Warszawski
www.mimuw.edu.pl/~dabrowski
r.dabrowski@mimuw.edu.pl

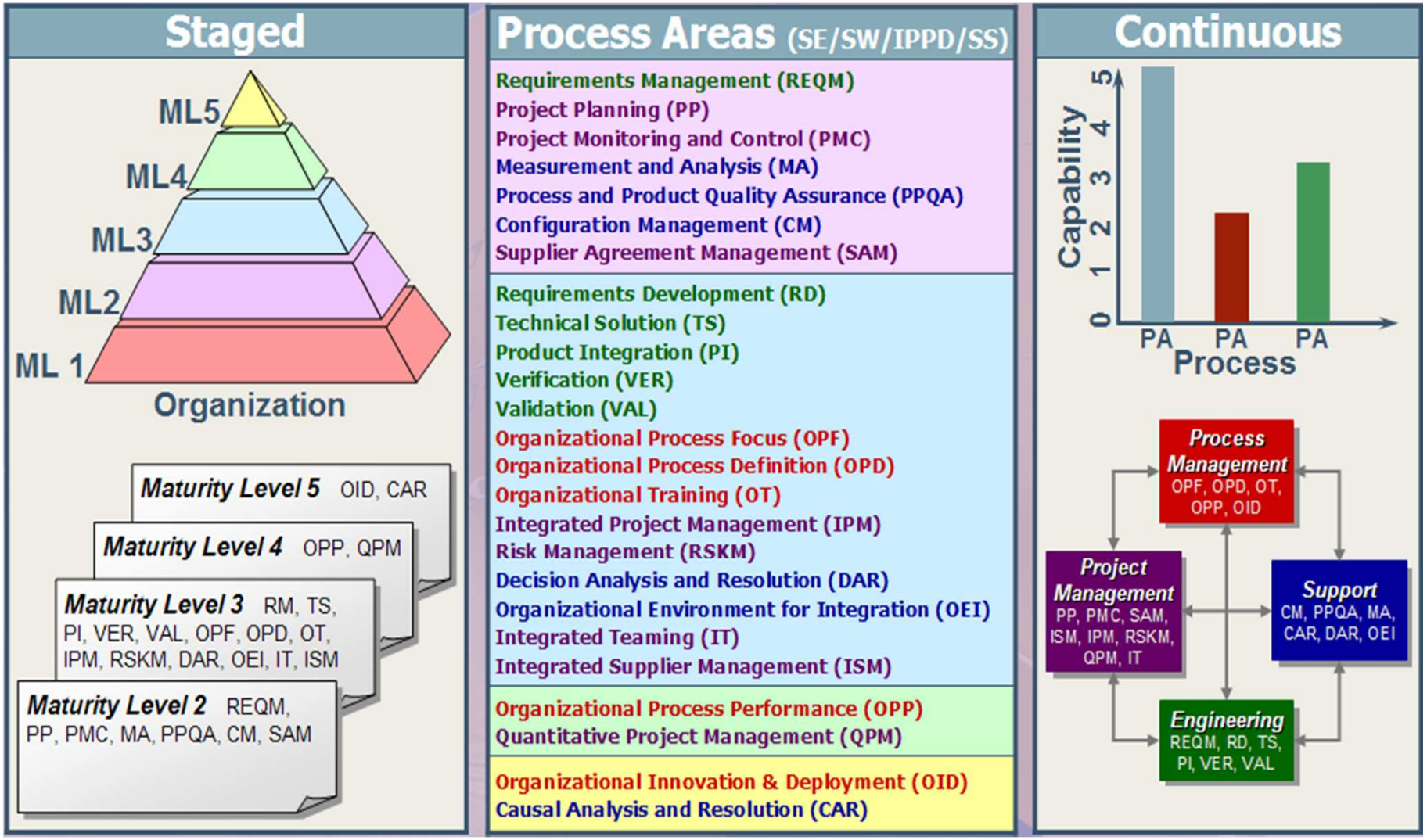
Trzy etapy nabywania nowych umiejętności

- Powtarzanie
  - hurra! – jedna procedura zadziałała
- Rozumienie
  - kiedy nie będzie działać?
  - jakie ma ograniczenia?
  - czy to właściwa procedura?
  - spróbujmy ją dostosować
- Biegłość
  - wiedza zintegrowana
  - pojedyncze techniki nie mają znaczenia

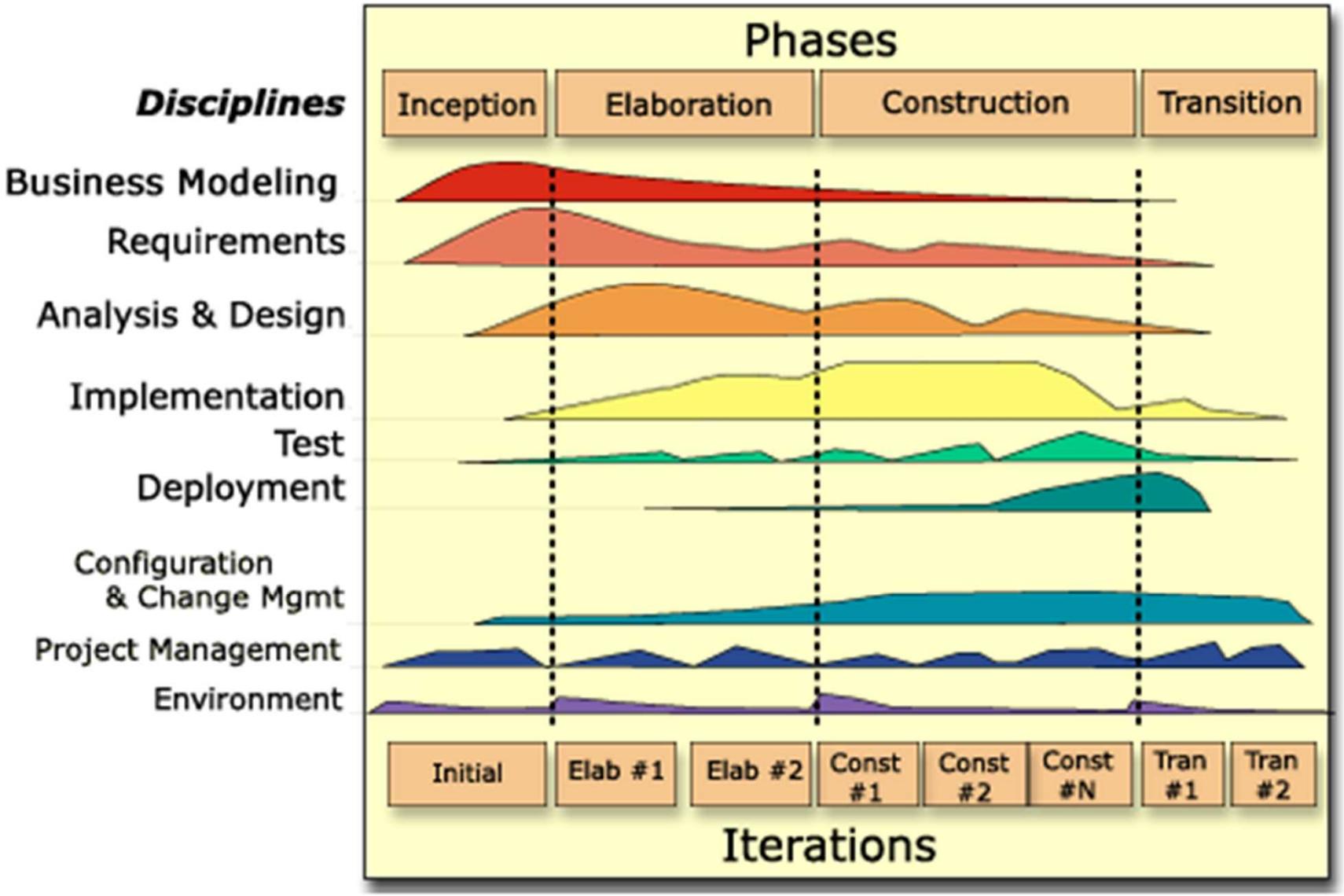
Trzy poziomy metodyk

- Metodyka
  - zestaw powiązanych praktyk / technik / metod / procesów / ...
- Poziom 1
  - bazowanie na procesach / standardach / szablonach
- Poziom 2
  - identyfikowanie technik praktycznie stosowalnych
- Poziom 3
  - pragmatyzm, pożyteczne pomysły
    (początkujący narzekaliby na brak wystarczających zasad)
- Należy unikać mieszania poziomów!

Capability Maturity Model Integration



Unified Process



Agile Software Development

- Manifesto for Agile Software Development
  - Individuals and interactions over processes and tools
  - Working software over comprehensive documentation
  - Customer collaboration over contract negotiation
  - Responding to change over following a plan

Principles behind the Agile Manifesto

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Principles behind the Agile Manifesto

7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity – the art of maximizing the amount of work not done – is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Metodyki adaptacyjne

- Zirytowanie podejściami nastawionymi na
  - dyscyplinę
  - kontrolowanie procedur
- Cele
  - odchudzenie procesów wytwarzania oprogramowania
  - zachowanie wysokiej jakości

- Powstały „lekkie” metodyki rozwoju oprogramowania
  - adaptacyjne / giętkie / elastyczne (ang. *agile*)
  - pierwowzorem jest Programowanie Ekstremalne (ang. *XP*) – Kent Beck, 1996-1999, pracował w firmie Chrysler nad oprogramowaniem przetwarzającym listy płac dla 87000 pracowników
  - popularny SCRUM – Ken Schwaber, firma Advanced Development Methods; Jeff Sutherland, firma Easel Corporation, 1995 konferencja OOPSLA

Procesy?

- Co??? Procesy???
- Po co mi (deweloperowi) procesy?
  - Przecież procesy są ZŁE!
  - Pracowałem w Wielkiej Korporacji i wiem, że są do niczego!
  - Nie mogę po prostu kodować? Tak jak lubię? ☠💣✳☥♎◆...
- Przecież to nie mój (dewelopera) problem, prawda?
  - Nie mogę zostawić procesów, polityki i planowania menedżerom?
  - Ja chcę po prostu kodować!
- Możesz to (deweloperze) zignorować, jeśli chcesz...
  - ... kodować rzeczy, których nikt nie używa
  - ... kodować rzeczy, których ludzie nienawidzą, a używają bo są zmuszeni
  - ... regularnie pracować po 60+ godzin w tygodniu / po nocach
  - ... krzyczeć na innych (deweloperów), zrzucać na nich winę
  - ... sam/a pracować na słabym kodzie (cudzym)
  - ...

Procesy!

- A jednak!
  - Rozumienie co należy zrobić i w jaki sposób
    - (bez ciągłego wynajdywania koła)
  - Umiejętność powtarzania dobrych pomysłów poprzez powielanie prostych, ustalonych wzorców
    - (bez konieczności zbytniego zastanawiania się nad nimi)
  - Umiejętność unikania powszechnych błędów
    - (znów – bez konieczności nadmiernego obciążania naszych cennych umysłów)

- Dobry proces pomaga:
  - Pracować efektywnie
  - Uniknąć kosztownych błędów
  - Określić co jest ważne, a co nie
  - Świadomie i z pełnym przekonaniem podejmować decyzje

Procesy

- Sama wydajność nie wystarczy
  - Zasada, że wszyscy pracują dla jednego celu, tak wydajnie jak potrafią jest oczywiście ważnym fundamentem
  - Ale co jeśli cel się zmieni? Lub jest częściowo nieznany? Lub w połowie drogi okaże się, że początkowy pomysł jest beznadziejny?
- Ale nie łudźmy się!
  - Procesy nie rozwiążą wszystkich problemów
  - W szczególności nie zmienią głupców w geniuszy
- Natomiast
  - Brak jasnych zasad (procesów) w nietrywialnym projekcie może projekt zabić
  - ... nawet jeśli w zespole są sami geniusze!

Procesy

- Tradycyjne metodyki były często ciężkie i nazbyt rygorystyczne
  - Mnóstwo zasad
  - Trudno zmienić proces
  - Trudno zmienić specyfikację produktu
- i dlatego powstały metodyki „lekkie”, „adaptacyjne”
- Ale
  - Jeśli funkcjonowanie procesu nie jest kontrolowane, nawet dobry proces będzie stawał się coraz cięższy, aż zmieni pracę w koszmar
- Trzeba unikać ponownego wynajdywania koła – rozpocząć od zasad, które się sprawdziły i dostosowywać je stopniowo do własnych potrzeb
- Dobry, lekki proces koncentruje się na kilku prostych zasadach i dużej ilości zdrowego rozsądku

Procesy

- Procesy o różnej „lekkości”
  - *Cowboy coding | Marine corps mentality*
    - Brak zasad, może zadziałać dla 1–3 deweloperów w małym projekcie.
  - Extreme Programming (XP)
    - Dużo prostych zasad, najlepszych praktyk. Lekki ale ścisły! Sprawdzi się w małych zespołach.
  - Scrum
    - Metodyka procesowa, dużo przydatnych wzorców, często stanowi rozszerzenie XP. Sprawdzi się w średnich zespołach.
  - Rational Unified Process
    - Dużo zasad, ról, wzorców. Sprawdzi się w dużych projektach.
  - CMMI
    - Kompletna metodyka. Największe projekty, duże organizacje.

Praktyki / zalecenia podejścia adaptacyjnego

1. Zespół, udziałowcy

- Przykładowy projekt
  - 1 kierownik
  - 1 architekt
  - 8 deweloperów
  - 2 praktykantów

- Mały projekt? Uwaga – zazwyczaj to nie wszyscy!
  - dyrektor działu, graficy, sprzedawcy, marketing, QA ...
  - zespół wsparcia
  - setki / tysiące (!) użytkowników

- Nawet średniej wielkości projekty mają wielu udziałowców
  - Każdy udziałowiec chce czegoś innego i na wczoraj...
  - Nie można uszczęśliwić każdego...
  - Ale trzeba spróbować ;-)

2. Role

- Najważniejsze role osób z zespołu:
  - deweloperzy
  - klient
- Klient jest członkiem zespołu
  - musi pracować razem z deweloperami
    - (w jednym pomieszczeniu)
  - może nie występować w tej roli osobiście
    - (przedstawiciel klienta)
- Role pomocnicze
  - *tester*
    - napisanie skryptów testowych na podstawie rozmów z klientem
  - *coach*
    - pomaga rozwiązywać napotkane problemy
  - *tracker*
    - zbiera statystyki dotyczące wykonanych zadań / czasu pracy
    - tworzy podsumowania postępów projektu

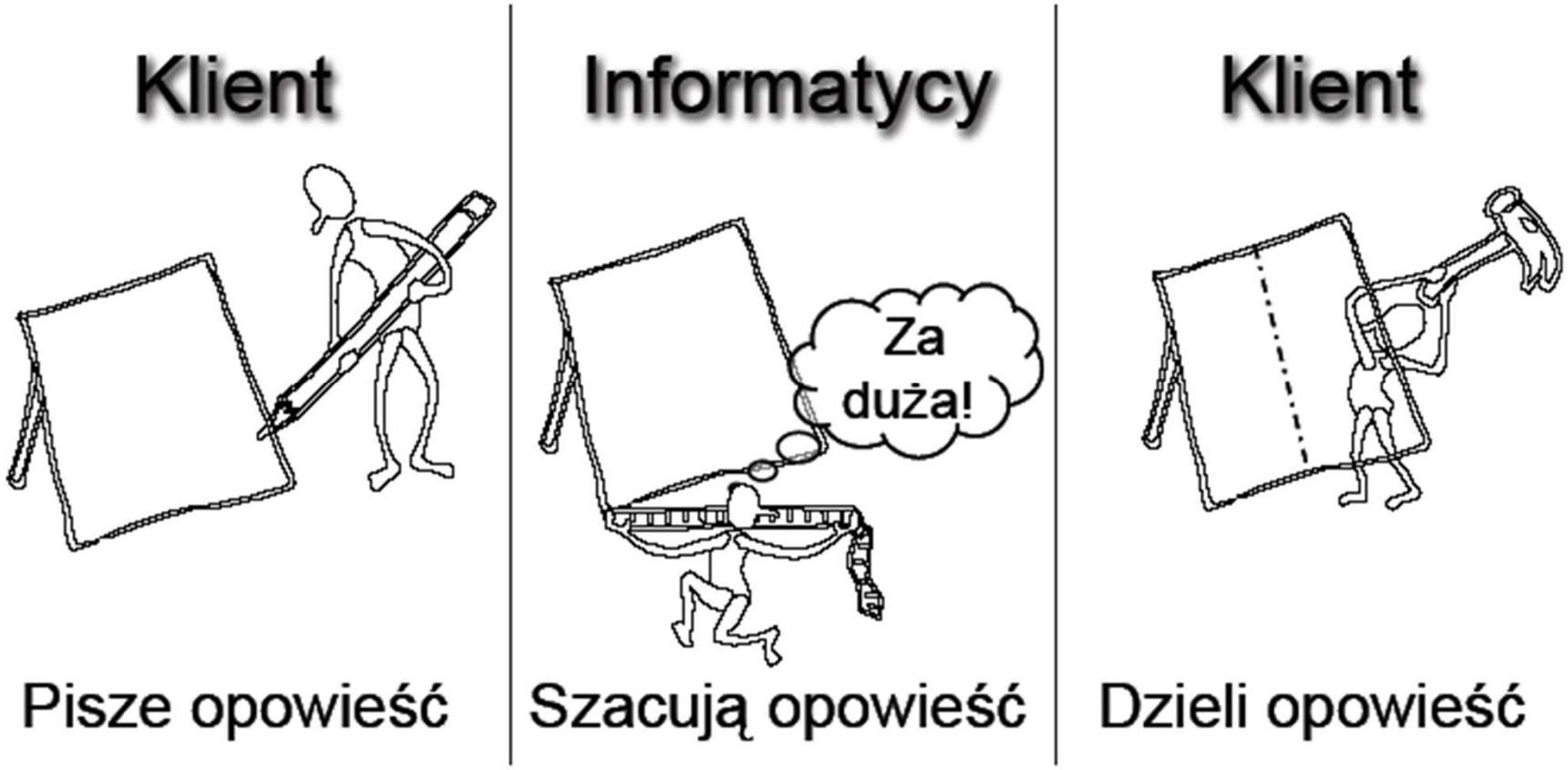
3. Relacje z klientem

- Współpraca zespołu z klientem
  - Zaufanie członków zespołu do klienta
  - Zaufanie klienta do członków zespołu
- Przedstawiciel klienta ciągłym źródłem wymagań
  - Wymagania dyskutowane na bieżąco
  - Ślad z dyskusji = „opowieści użytkowników”
  - Każda opowieść jest zapisana w kilku zdaniach
    - („na jednej kartce papieru”)
  - Może być oznaczona dodatkowymi atrybutami
    - (data utworzenia, typ, numer)
    - ułatwią priorytetyzację
- Opowieść użytkownika nie jest kompletnym wymaganiem
  - wymagania są jedynie przekazywane w bezpośredniej rozmowie
- Po co zapisywać opowieści użytkownika?
  - można uporządkować rozmowę o wymaganiach
  - łatwo przydzielać funkcjonalność do poszczególnych wydań
  - można śledzić postęp projektu
- Ważne, aby każda opowieść miała wartość dla klienta i była testowalna!

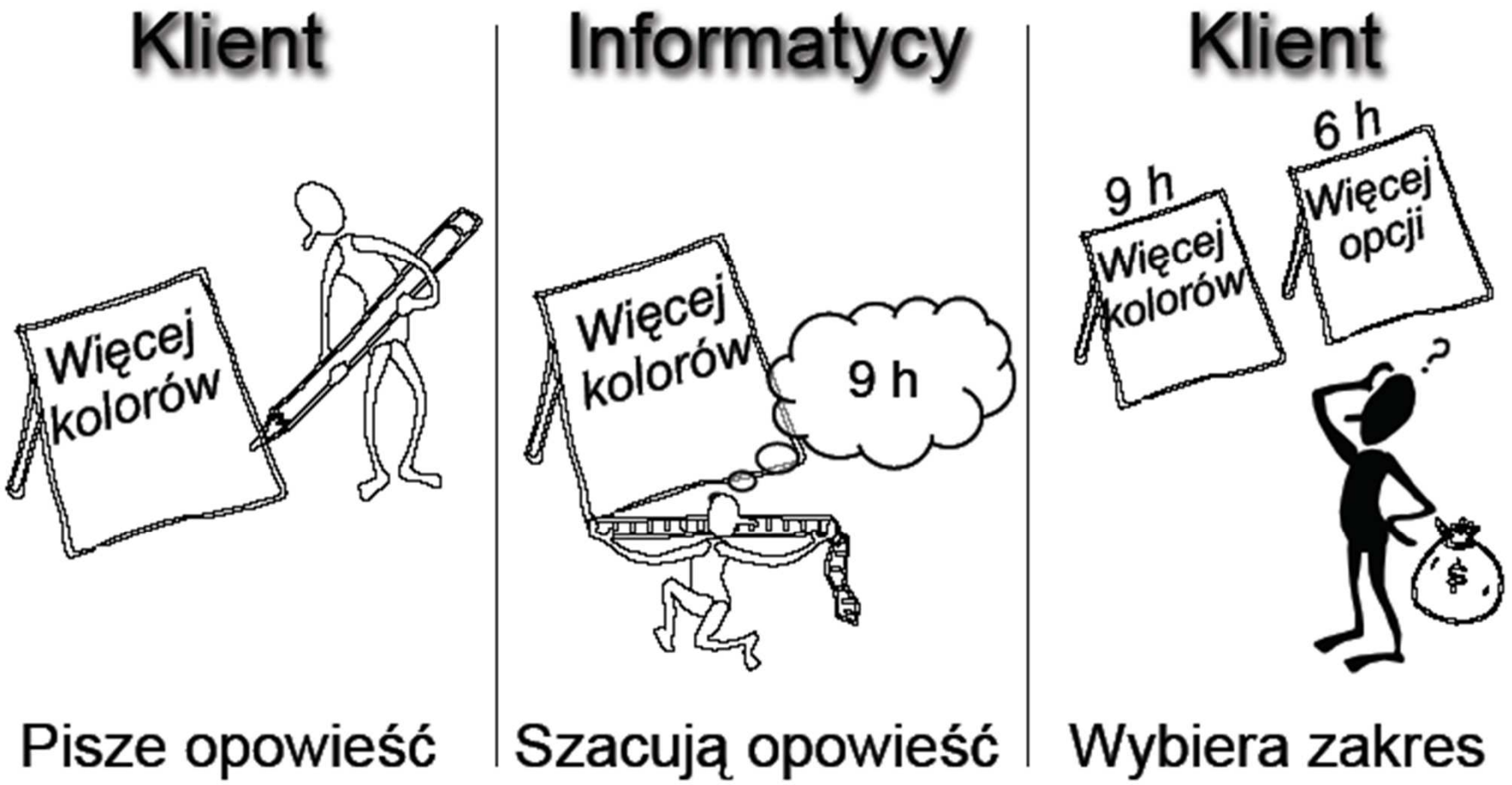
4. Metafory

- Wyjaśnianie działania systemu za pomocą „metafor” (w terminach zrozumiałych dla klienta)
- Metafora przydaje się zwłaszcza do ukrycia terminów technicznych
  - np.
    - „komputerowy segregator z fakturami”
  - zamiast
    - „relacja w bazie danych przechowująca dane faktur”

5. Opowieści użytkowników



6. Gra planistyczna



6. Gra planistyczna

- Klient opowiada „opowieści użytkownika”
- Opowieści są szacowane metodą „Gra Planistyczna”
  - (regularnie, np. cotygodniowo)

- Opowieści są dzielone na jednostki
- Jednostki mierzą złożoność, nie czas wykonania
- Opowieści duże są dzielone na mniejsze

- Klient priorytetyzuje pracę
  - (na podstawie informacji o stopniu skomplikowania)

- Ludziom lepiej idzie oszacowywanie złożoności niż czasu wykonania
- Małe zadania są bardziej przewidywalne
- Klient zna i kontroluje koszty funkcjonalności

7. Planowanie

- „Plany są niczym, planowanie wszystkim”

  [powiedzenie ludowe]

- Ciągle prawdziwe
  - Planowanie jest kosztem
  - Plany szybko się przeterminowują
  - Niektórych rzeczy nie da się zaplanować

- Ale bez planowania...
  - Zobowiążemy się zrobić więcej niż jesteśmy w stanie
  - O niektórych elementach zupełnie zapomnimy
  - Najpierw zrobimy rzeczy fajne, rzeczy ważne / pilne będziemy robić nocami
  - Niektóre zadania będziemy robić wielokrotnie lub w nieoptymalnej kolejności

7. Planowanie

- Krok 1: *Collect*
- Krok 2: *Prioritize*
- Krok 3: *Estimate*
- Krok 4: *Reorder*
- Krok 5: *Specify* (ale nie za bardzo!)

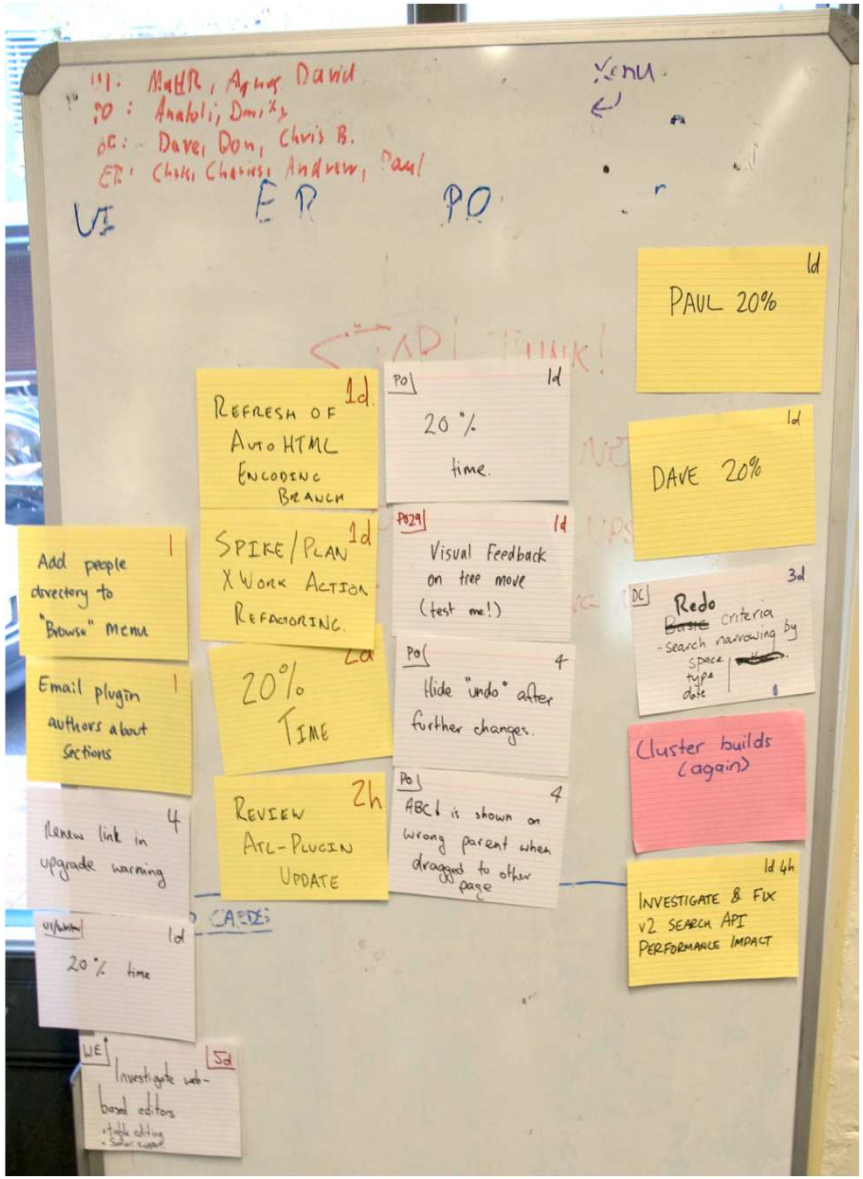
- Estymacje
  - Estymacje pracy w jednostkach względnych (nie ma konieczności szacowania w dniach roboczych!)
  - W estymacje zaangażowani różni deweloperzy, wszyscy którzy będą dany element kodować
  - Jednostki planowania: 1,2,4,8,16 lub 1,2,3,5,8,13,21
    - (nie chodzi o precyzję!)
  - Rozbijanie dużych zadań na mniejsze (<=3)

7. Planowanie

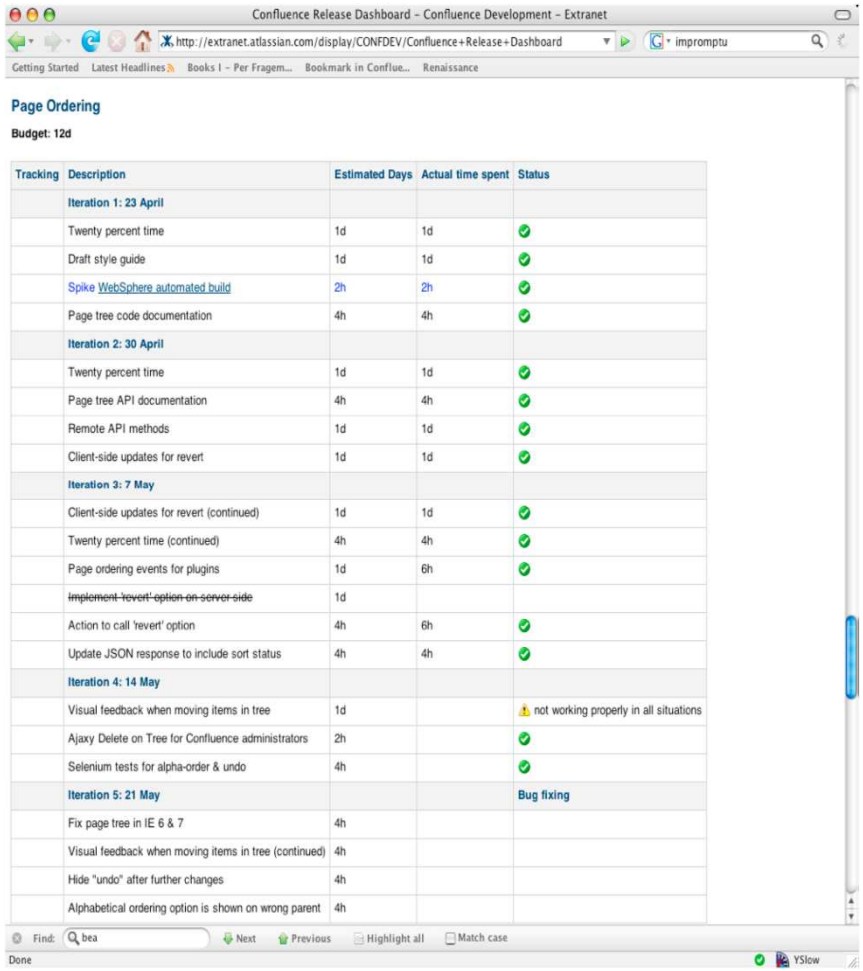
- System punktów:
  - 1: Wiem dokładnie jak to zrobić, zrobię to w pół dnia
  - 2: Wiem dokładnie jak to zrobić, trochę to zajmie
  - 3: Jakoś to zrobimy
- Często 3 punkty przekształcają się w większą pracę, niż przewidywana – na takie (lub większe) opowieści trzeba uważać.
- Niewielkim wysiłkiem można zaplanować prace na kolejne 2–3 miesiące
- Prace są realizowane w podejściu „top-down”, plany dostosowywane w trakcie prac
- Zmiana planów nie jest kosztowna (bo nie poświęciliśmy na nie dużo czasu)

8. Monitorowanie

- Plany tygodniowe
- Np. na tablicy



8. Monitorowanie

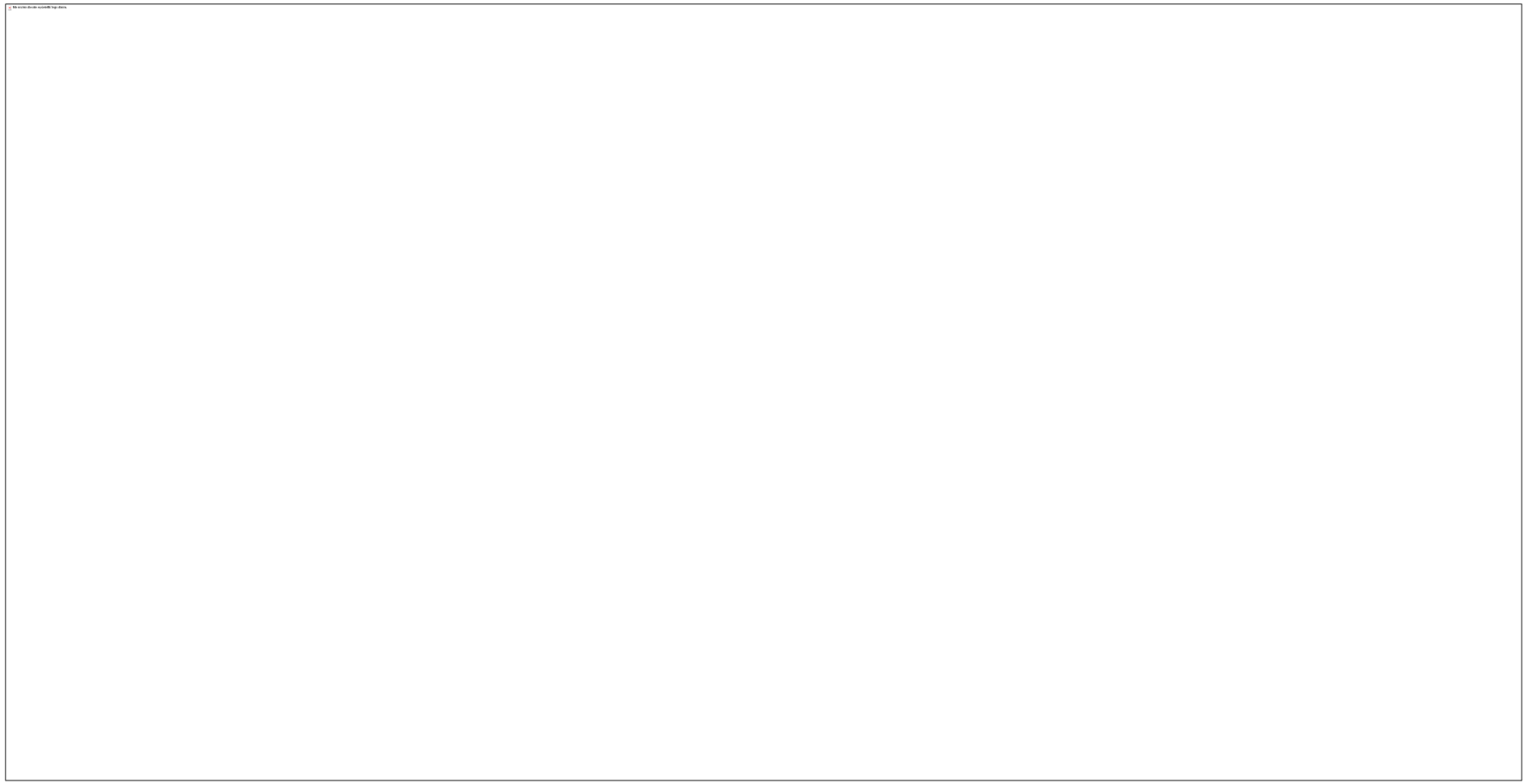


- Plany wydań (np. dwumiesięczne)
- Np. w wiki

8. Monitorowanie

- Stałe monitorowanie postępu
  - Weryfikacja oszacowań i szybkości realizacji prac
- Sprawdzanie zasadności oszacowania pracy pozostającej do wykonania
  - czy ciągle trzeba ją wykonać?
  - czy jest zaplanowana we właściwej kolejności?
- Wczesne podejmowanie decyzji
  - od razu rezygnujemy z cech jeśli widzimy, że nie da się ich uzyskać (w tym wydaniu)
- *„Rethink the plan"*

9. Wypalanie



10. Iteracyjność

- Zazwyczaj oznacza:
  - „cyklicznie i przyrostowo”
- Cyklicznie:
  - analiza, planowanie, implementacja, testy, zebranie uwag
- Przyrostowo:
  - Większość cech dostarczana etapowo, nie ma „wielkiego wybuchu”
- Przykład:
  - Co 2 tygodnie wydanie pośrednie, po nim testy i wdrożenie, na koniec zebranie uwag (uwzględnianych w planach następnej iteracji)

11. Elastyczne podejście do zmian

- Akceptujemy – nawet mile witamy! – zmiany

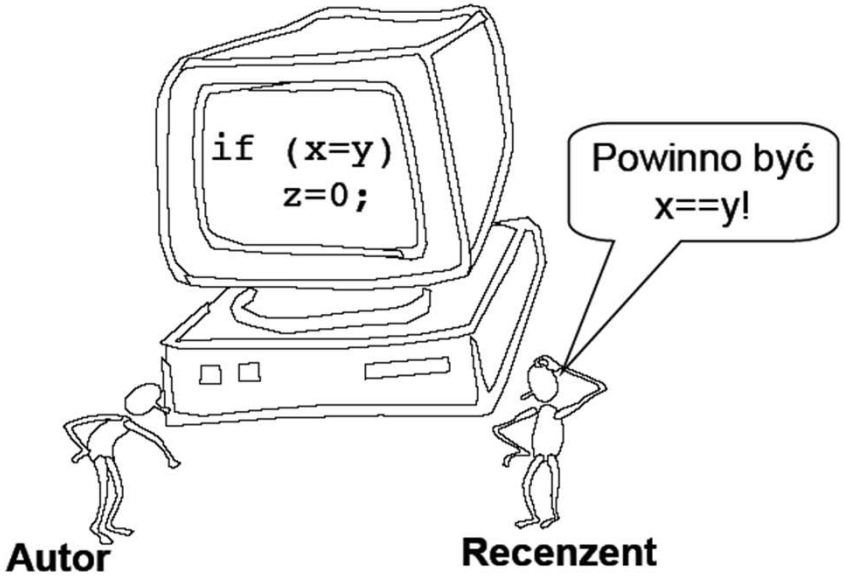
- Nie spędzamy zbyt wiele czasu na planowaniu na początku projektu
- Zamiast tego stale wykonujemy re-planowanie w trakcie trwania projektu

- Akceptujemy, że fragmenty systemu będą rozszerzane / przepisywane (dodatkowy koszt)
- Oszczędzamy na przesadnie szczegółowym projektowaniu

11. Elastyczne podejście do zmian

- Podejście bazujące na dobrej współpracy z klientem
  - klient w dowolnym momencie może zmienić zdanie
    - zaproponować zmianę wymagań
  - nie ma na początku dokładnego kontraktu
    - określającego pełen zakres działań oraz koszt projektu
  - klient płaci na bieżąco za wykonaną pracę
    - zdaje sobie sprawę z tego, że zmiana elementów już zaimplementowanych będzie go kosztowała dodatkowo
  - deweloperzy nie muszą się martwić reworkami
    - zawsze dostaną wynagrodzenie za swoją pracę

12. Przeglądy kodu



- Wspólny standard kodowania dla całego zespołu
  - bez tego nie da się wygodnie pracować w parach
- Kod jest własnością całego zespołu
  - każdy zna większość kodu
- System zarządzania wersjami
  - podstawa!!!
- Otwarta przestrzeń pracy dla zespołu
  - Komunikacja
- Stałe przeglądu kodu
  - (np. programowanie w parach)

13. Współwłasność kodu

- Zapobiega „silosom wiedzy”
- Każdy (posiadający elementarne kompetencje) może zmieniać dowolny fragment kodu
  - brak restrykcji w systemie kontroli wersji
- Brzmi groźnie?
  - Nie tak bardzo, jeśli mamy dobre i zautomatyzowane testy
- Bardziej elastyczne podejście, nie ma wąskich gardeł
  - „*Makes worklife more fun too*”

14. Stała integracja produktów

- Każdy wykonuje pełen zestaw testów przed przekazaniem kodu (*commit*)
  - (najlepiej automatycznych)
- Stała kontrola linii głównej (*trunk*), cykliczne automatyczne budowanie (*build*) i testowanie
- Wczesne wykrywanie problemów, łatwiej je naprawić
- Przez cały czas aplikacja jest utrzymywana w stanie umożliwiającym jej uruchomienie

15. Małe wydania

- Wiele małych wydań!
  - Np.:
  - Duże (*major*) wydanie (*release*) co kwartał
  - Małe (*minor*) wydanie (*release*) co miesiąc
- Wymagania (także narzędzia, użytkownicy, deweloperzy, …) zmieniają się zbyt szybko, by planować na rok naprzód
- Wcześnie otrzymujemy opinie klientów /użytkowników
- Deweloperzy cieszą się widząc działającą wersję!!!
- Krótki cykl kodowanie-debugowanie – dużo bardziej efektywny

16. Komunikacja

- Najważniejsze we wszystkich metodykach adaptacyjnych
- Jak zabić projekt? Zapobiec komunikacji.
  - Najważniejsze: rozmawiać! Żadna dokumentacja tego nie zastąpi.
  - Oczywiście trzeba pisać maile, uzupełniać wiki, postować na forum, korzystać z komunikatórów / telefonów, itp..
  - Ale jeśli tylko możliwe – trzeba „pogawędzić”
  - (Podobno nasze geny aż tak się nie zmieniły od epoki kamiennej)
- Regularne spotkania
  - Nieformalne pogawędki, spotkania ad-hoc – bardzo ważne, ale łatwo przegapić dużo ważnych spraw
  - Warto ustalić regularne spotkania o podobnym przebiegu
  - *„Meetings take time - not having them takes more time!"*
  - Konieczne: jasna agenda spotkania + moderator

16. Komunikacja

- A: *Stand-up meeting*
  - Codziennie o 10:00
  - Każdy ma minutę na opowiedzenie co robił wczoraj i co będzie robił dziś
  - Jeśli pojawia się temat do dyskusji – jest odkładany do zakończenia spotkania, kontynuowany „off-line”
  - Najlepiej: 15 minut

16. Komunikacja

- B: *Planning meeting*
  - Co tydzień, około 45 minut
  - Uczestniczą wszyscy deweloperzy pracujący nad wydaniem
  - Przedstawiciele zespołów prezentują postęp zespołów i plany na kolejny tydzień
  - Dyskusja / zmiana planów
  - Każdy zespół może odmówić dodatkowej pracy
    - *„No overcommitment!"*

16. Komunikacja

- C: *Retrospective meeting*
  - Co 2-4 tygodnie, około 90 minut
  - Agenda:
    - Co nam przeszkadza w procesie?
    - Jak usprawnić nasze zespoły i miejsce pracy?
    - Co nie poszło dobrze od ostatniego spotkania?
    - Co poszło świetnie?
  - Zbieramy wnioski na whiteboard-zie
    - Dobre rzeczy po lewej
    - Słabe po prawej
    - Każdy oddaje głosy (5 głosów max)
    - Dyskutujemy 3-4 najważniejsze zagadnienia
  - Ustalamy działania, przypisujemy osoby, follow-up przy następnym spotkaniu

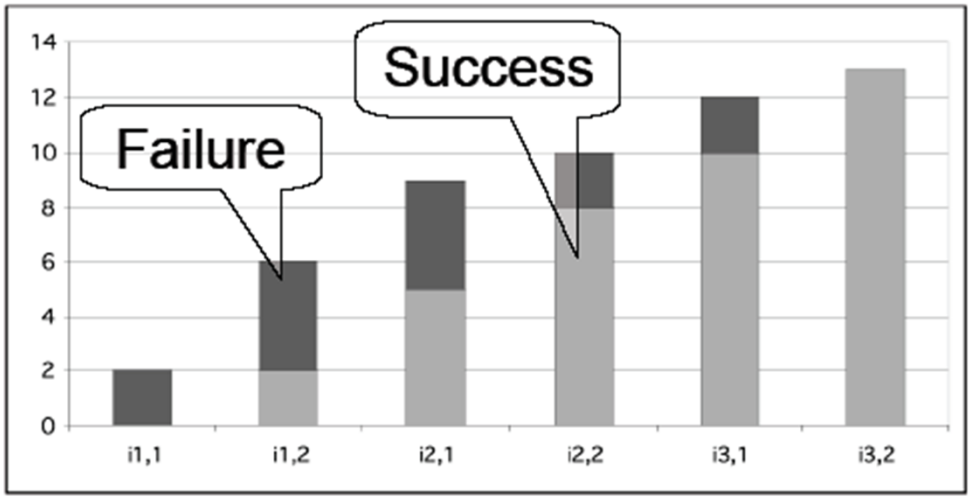
16. Komunikacja

- Inne spotkania
  - Tylko gdy potrzebne
  - Tylko z koniecznymi uczestnikami
  - Tylko z jasną agendą
  - Przykład
    - Spotkanie z Alą i Bartkiem czy przejść z Hibernate 2.0 na 3.0 aby poprawić wydajność

17. Testy komponentów (unit testing)

- O tym już było…
- Najlepiej automatyzowane

18. Testy akceptacyjne



- Testy pochodzące od klienta
- Klient określa, jak system musi się zachować w określonych warunkach
  - Najlepiej gdy testy mogą być wykonywane automatycznie
- Zalety
  - klient jest przekonany, że system spełnia jego wymagania
  - testując na bieżąco jesteśmy w stanie powiedzieć, jak wygląda postęp projektu (ile % funkcjonalności zostało poprawnie zaimplementowane)
- Obserwując zmianę w czasie, widać że kolejne partie systemu zostały zaimplementowane.
- Słupki stale rosną w czasie, gdyż wraz z przyrostem funkcjonalności, przyrastają przypadki testowe

19. Zapewnianie jakości

- Dbaj o prostotę
- Unikaj (niepotrzebnej) optymalizacji
- Dla każdej jednostki kodu opracuj najpierw zestaw testów, potem napisz kod
- Automatyczne wykonanie testów
- Refaktoryzacja

- Różnica względem metodyk tradycyjnych:
  - optymalizować kod należy tylko wtedy, gdy jest to konieczne (nie próbujemy przewidywać problemów, stawiamy na proste rozwiązania i refaktoring)
  - przypadki testowe należy przygotować przed rozpoczęciem kodowania
  - testy wykonywane automatycznie, na bieżąco wychwytywane błędy
  - stałe poprawianie czytelności kodu (czyli znów refaktoryzacja)

19. Zapewnianie jakości

- Kod musi przejść wszystkie testy jednostkowe zanim przekażesz go do eksploatacji
- Dla każdego wykrytego błędu (na przetestowanym kodzie) utwórz dodatkowy zestaw testów
- Często integruj kod, wykonuj testy integracyjne
- Często wykonuj testy akceptacyjne, publikuj ich wyniki

Podsumowanie

- Metodyki adaptacyjne nie są rozwiązaniem uniwersalnym
  - Klient przez cały czas pracuje z zespołem?
    - A kto robi to, co robił do tej pory?
  - Brak dokumentacji?
    - A jeśli po pewnym czasie trzeba wrócić do prac nad starym modułem?
  - Brak fazy projektowania?
    - Czy rzeczywiście potrafimy robić refaktoring?
  - Brak długoterminowego planowania?
    - Kto mi powie kiedy system będzie ukończony?

Przypomnienie: IO = U.P. >> A.S.Dev. = Agile Unified Process