Jarosław Bochenek
email: jjbochenek@wsb-nlu.edu.pl
ORCID: 0009-0000-3499-8313
Wyższa Szkoła Biznesu – National Louis University

Architektoniczny dług technologiczny. Ustrukturyzowany przegląd literatury w zakresie detekcji i refaktoryzacji

Architectural Technical Debt. A Structured Literature Review on Detection and Refactoring

DOI: 10.66935/978-83-65926-17-3.17

Cytuj jako:
Bochenek, J. (2025). Architektoniczny dług technologiczny. Ustrukturyzowany przegląd literatury w zakresie detekcji i refaktoryzacji. W: J. Lizak, M. W. Sidor (red.), *Nowe wyzwania w naukach społecznych i technicznych: podejście interdyscyplinarne: T. 2* (s. 1-). Wydawnictwo WSB-NLU.

Streszczenie: W pracy podjęto problematykę architektonicznego długu technologicznego (ATD), wynikającego z suboptymalnych decyzji projektowych. Zjawisko to prowadzi do degradacji systemów i drastycznego wzrostu kosztów ich utrzymania, które mogą pochłonąć nawet do 70% wydatków całego cyklu życia oprogramowania. W przeciwieństwie do klasycznego długu w kodzie, ATD wpływa głównie na wewnętrzne cechy oprogramowania, utrudniając jego ewolucję. Badanie przeprowadzono metodą ustrukturyzowanego przeglądu literatury, szczegółowo analizując 15 publikacji z bazy Scopus. Zidentyfikowano, że do skutecznej detekcji ATD kluczowe jest wykorzystanie zaawansowanych metryk (takich jak algorytm PageRank czy macierze DSM) oraz analiza zapachów architektonicznych, w tym zależności cyklicznych, niestabilnych i typu piasta. Główną tezą pracy jest to, że optymalna mitygacja długu wymaga systematycznej refaktoryzacji opartej na rygorystycznym stosowaniu zasad projektowych (szczególnie zasady odwrócenia zależności z paradygmatu SOLID) oraz głębokiej dekompozycji systemu. Artykuł dostarcza brakującego, syntetycznego powiązania wpływu zapachów architektonicznych z konkretnymi metodami priorytetyzacji refaktoryzacji. Oryginalnym wkładem pracy jest jednak krytyczna ocena tego zmechanizowanego podejścia i uwypuklenie socjotechnicznego oraz biznesowego kontekstu ewolucji oprogramowania.

Abstract: The paper addresses the issue of architectural technical debt (ATD) resulting from suboptimal design decisions. This phenomenon leads to system degradation and a drastic increase in maintenance costs, which can consume up to 70% of the total software lifecycle expenditure. Unlike classic code debt, ATD mainly affects the internal characteristics of software, hindering its evolution. The study was conducted using a structured literature review, analyzing 15 publications from the Scopus database in detail. It was identified that the use of advanced metrics (such as the PageRank algorithm or DSM matrices) and the analysis of architectural smells, including cyclic, unstable, and hub dependencies, are key to the effective detection of ATD. The main thesis of the paper is that optimal debt mitigation requires systematic refactoring based on the rigorous application of design principles (especially the inversion of dependencies principle from the SOLID paradigm) and deep system decomposition. Article provides a missing, synthetic link between the impact of architectural smells and specific refactoring prioritization methods. However, the original contribution of the work is a critical assessment of this mechanized approach and an emphasis on the socio-technical and business context of software evolution.

Słowa kluczowe: architektoniczny dług technologiczny, refaktoryzacja, zapachy architektoniczne, architektura oprogramowania

Keywords: architectural technical debt, refactoring, architectural smells, software architecture

Wprowadzenie

Współczesna inżynieria oprogramowania wymaga ciągłego balansowania pomiędzy szybkością dostarczania nowych funkcjonalności a utrzymaniem wysokiej jakości strukturalnej systemów. W tym kontekście kluczowym wyzwaniem staje się architektoniczny dług technologiczny (Architectural Technical Debt – ATD). W przeciwieństwie do długu na poziomie kodu, ATD wpływa głównie na wewnętrzne cechy systemu, takie jak jego łatwość utrzymania oraz zdolność do dalszej ewolucji. Często nie jest on zauważany przez użytkowników systemu, ale ma wpływ na jego utrzymanie oraz rozbudowę.

Mimo że w literaturze przedmiotu można odnaleźć obszerne przeglądy dotyczące długu architektonicznego, większość z nich ogranicza się do ogólnego mapowania definicji oraz kategoryzacji narzędzi detekcyjnych. Brakuje w nich syntetycznego powiązania wpływu poszczególnych zapachów architektonicznych z konkretnymi metodami priorytetyzacji refaktoryzacji, w szczególności tymi opartymi na zasadach SOLID. Oryginalnym wkładem niniejszego artykułu, stanowiącym jego specyficzną wartość dodaną na tle istniejących publikacji, jest wypełnienie tej luki oraz wyjście poza redukcjonistyczne, metryczne podejście. Praca oferuje krytyczną analizę strategii mitygacji ATD, uwypuklając

marginalizowany dotąd wymiar socjotechniczny oraz biznesowy. Zderza ona teoretyczne modele refaktoryzacji z wyzwaniami praktyki inżynierskiej, wskazując na nieuchronność kompromisów architektonicznych (takich jak centralizacja logiki) oraz konieczność uwzględnienia czynników kognitywnych w procesie zarządzania długiem.

Celem niniejszego artykułu jest przeprowadzenie ustrukturyzowanego przeglądu literatury w celu krytycznej analizy zjawiska architektonicznego długu technologicznego. Badanie koncentruje się na usystematyzowaniu definicji ATD w kontekście inżynierii oprogramowania, identyfikacji metod jego detekcji na poziomie struktury i kodu oraz ewaluacji rekomendowanych strategii jego spłacania poprzez refaktoryzację i przebudowę architektury. Realizacji tak określonego celu służy poszukiwanie odpowiedzi na następujące pytania badawcze:

- RQ1: Jak w literaturze z dziedziny inżynierii oprogramowania definiuje się architektoniczny dług technologiczny (ATD) w kontekście budowy, struktury i kodu źródłowego systemów informatycznych?
- RQ2: Jakie antywzorce projektowe oraz metryki kodu źródłowego są najczęściej wskazywane w literaturze jako narzędzia do detekcji długu architektonicznego?
- RQ3: Jakie metody prewencji i spłacania długu architektonicznego są rekomendowane w literaturze przedmiotu oraz jakie wyzwania i kompromisy (trade-offs) natury biznesowo-kognitywnej ograniczają ich praktyczną implementację w projektach informatycznych?

Metodologia przeglądu literatury

W celu identyfikacji relewantnych badań dotyczących długu architektonicznego, przeprowadzono ustrukturyzowany przegląd literatury. Jako źródło danych wybrano bazę naukową Scopus. Proces wyszukiwania został przeprowadzony przy użyciu zapytania, które obejmowało słowa kluczowe związane z długiem architektonicznym oraz inżynierią oprogramowania. Zastosowano następujące zapytanie wyszukiwawcze:
TITLE-ABS-KEY ("architectural technical debt" OR "architectural debt") AND TITLE-ABS-KEY (software OR system* OR code) AND (LIMIT-TO (SUBJAREA , "COMP")) AND (LIMIT-TO (LANGUAGE , "English"))

Aby zapewnić wysoką jakość i trafność wyników, wyszukiwanie zostało ograniczone do publikacji z obszaru informatyki (Computer Science) oraz napisanych w języku angielskim. Proces doboru literatury został podzielony na trzy główne etapy, co pozwoliło na stopniową i rygorystyczną filtrację wyników:

- Etap 1: Identyfikacja (Wyszukiwanie wstępne): Wykonanie zdefiniowanego zapytania w bazie Scopus zwróciło łącznie 109 artykułów.
- Etap 2: Wstępna selekcja (Przegląd tytułów i abstraktów): W tym kroku przeanalizowano tytuły oraz abstrakty wszystkich 109 zidentyfikowanych publikacji. Głównym kryterium włączenia na tym etapie było bezpośrednie odniesienie się do problematyki długu architektonicznego oraz zapachów architektonicznych w kontekście systemów oprogramowania. Publikacje niespełniające tego kryterium, będące poza głównym zakresem tematycznym badań, zostały odrzucone. W wyniku tej selekcji do dalszej analizy wybrano 21 artykułów.
- Etap 3: Szczegółowa selekcja (Analiza pełnych tekstów): Ostatnim krokiem była dogłębna analiza pełnych tekstów 21 wyselekcjonowanych publikacji. Oceniano je pod kątem wkładu wnoszonego w problematykę badawczą, jakości naukowej oraz szczegółowości opisu metod lub przypadków użycia. Po przeczytaniu pełnych treści ostatecznie zakwalifikowano 15 artykułów.

Ostateczna baza, na której opiera się niniejszy artykuł, składa się z 15 rygorystycznie wyselekcjonowanych pozycji literaturowych. Stanowią one fundamentalny trzon do dalszej analizy, syntezy wiedzy oraz wyciągania wniosków przedstawionych w kolejnych rozdziałach niniejszej pracy.

Definicja i geneza

Architektoniczny dług technologiczny w dziedzinie inżynierii oprogramowania jest definiowany jako kategoria długu technologicznego dotycząca „dużych” decyzji projektowych, stanowiąca zobowiązanie warunkowe, którego wpływ ogranicza się głównie do wewnętrznych cech systemu, takich jak łatwość utrzymania i ewolucji (Martini i in., 2018). Dług architektoniczny w systemach oprogramowania składa się z konstrukcji projektowych lub implementacyjnych, które są korzystne w perspektywie krótkoterminowej, ale tworzą kontekst techniczny, który może sprawić, że przyszłe zmiany będą bardziej kosztowne lub wręcz niemożliwe do zrealizowania. W związku z tym ATD to dług zaciągnięty na poziomie architektonicznym projektowania oprogramowania, czyli w decyzjach dotyczących wyboru struktury (np. warstwowania, podziału na podsystemy, interfejsów), wyboru technologii (np. frameworków, pakietów, bibliotek, podejścia do wdrażania), a nawet języków, procesu rozwoju oraz platformy. W przeciwieństwie do długu na poziomie kodu, który jest łatwo wykrywalny przez analizatory statyczne, ATD jest trudny do zidentyfikowania, wiąże się z bardzo zróżnicowanymi kosztami naprawczymi i często

bywa świadomie unikany przez zespoły ze względu na swój przytłaczający charakter. (Verdecchia i in., 2021).

Występowanie architektonicznego długu technologicznego jest w wielu przypadkach zjawiskiem narastającym wraz z rozwojem i ewolucją systemów informatycznych. W miarę jak systemy rosną, a ich cykl życia wydłuża się do wielu lat, pierwotne decyzje projektowe stają się ograniczeniami, które hamują, a nawet uniemożliwiają przyszłą ewolucję. Prowadzi to do degradacji architektury, objawiającej się poprzez jej erozję (naruszanie pierwotnych założeń) lub dryf (rozmywanie zasad architektonicznych, co ułatwia powstawanie naruszeń). Degradacja jakości architektury oprogramowania wynika między innymi z braku czasu lub stosowania tzw. skrótów w kodowaniu, co prowadzi do naruszenia kluczowych zasad projektowania oprogramowania, takich jak chociażby modularność. Skutkuje to znacznym wzrostem kosztów utrzymania, które w skrajnych przypadkach mogą pochłonąć od 50% do nawet 70% całkowitego kosztu cyklu życia systemu (Maikantis i in., 2021).

Nieodpowiednie decyzje projektowe, na przykład utworzenie scentralizowanego komponentu obsługującego większość procesów systemu, wprowadzają "architektoniczne zapachy" (architectural smells), negatywnie wpływając na jakość i łatwość utrzymania systemu. Te zapachy manifestują się jako niepożądane zależności, niezrównoważona dystrybucja obowiązków i nadmierne sprzężenie między komponentami; ze względu na łamanie dobrych praktyk projektowych są one powszechnie uważane za jedną z najczęstszych form architektonicznego długu technologicznego. Niewykryte zapachy architektoniczne "przeciekają" do kolejnych etapów projektowania i implementacji, a koszt ich usunięcia drastycznie rośnie w późniejszych fazach cyklu życia oprogramowania, szczególnie w dużych projektach przemysłowych (Mumtaz i in., 2021). W celu rozwijania systemu, w którym nagromadził się dług, deweloperzy często stosują obejścia (workarounds) oraz skomplikowane rozwiązania, co wprowadza problemy z jakością i generuje kolejne problemy (MacCormack i Sturtevant, 2016; Verdecchia i in., 2021).

W literaturze przedmiotu najczęściej wskazywanymi metodami do detekcji długu architektonicznego są analiza kodu źródłowego oraz wspomniane zapachy architektoniczne, które stanowią mierzalne przejawy suboptymalnych decyzji projektowych wpływających na strukturę całego systemu (Sousa i in., 2023). Do najpowszechniej identyfikowanych zapachów architektonicznych należą: zależności cykliczne (Cyclic Dependency), zależności typu piasta (Hub-like Dependency) oraz zależności niestabilne (Unstable Dependency) (Bochicchio i in., 2025). W celu syntetycznego ujęcia problematyki, w tabeli 1 zestawiono

charakterystykę, sposoby pomiaru oraz priorytetyzację spłaty dla trzech najpowszechniej identyfikowanych zapachów architektonicznych.

Tabela 1

Charakterystyka, miary i koszty mitygacji kluczowych zapachów architektonicznych

Typ zapachu	Charakterystyka	Miara krytyczności	Koszty spłaty
Zależności cykliczne	Sytuacja, w której komponenty tworzą zamkniętą pętlę powiązań (np. A zależy od B, a B od A). Naruszają acykliczność struktury modułów, co uniemożliwia ich samodzielne rozwijanie i testowanie.	Mierzona liczbą elementów współdzielących cykl. Należy jednak ocenić kontekst, gdyż część z nich bywa uznawana za konieczne i naturalne powiązania (np. wywołania zwrotne).	Najbardziej kosztowna refaktoryzacja (dziesiątki/setki roboczogodzin) generująca najwięcej skutków ubocznych.
Zależności typu piasta	Abstrakcje o skrajnie dużej liczbie połączeń wchodzących i wychodzących. Jeden komponent jest powiązany z nadmierną liczbą innych elementów.	Mierzona liczbą połączeń wchodzących i wychodzących.	Umiarkowany/niski nakład pracy i brak skutków ubocznych. Rekomenduje się najwyższy priorytet ich usuwania ze względu na najlepszy stosunek niskich kosztów naprawy do wysokich korzyści.
Zależności niestabilne	Występują, gdy komponent o wysokim potencjale zmian jest fundamentem dla modułów bardziej stabilnych. Grozi to efektem domina (efektem fali) przy	Mierzona liczbą pakietów powodujących naruszenie stabilności struktury.	Często postrzegane jako najmniej przydatne do refaktoryzacji, ponieważ bywają wynikiem celowego zastosowania sprawdzonych wzorców

Typ zapachu	Charakterystyka	Miara krytyczności	Koszty spłaty
	modyfikacjach.		projektowych.

Źródło: Opracowanie własne na podstawie (Bochicchio i in., 2025).

Innym przykładem zapachu architektonicznego jest tzw. God Component, charakteryzujący się nadmierną koncentracją odpowiedzialności oraz niewystarczającą modularyzacją, która utrudnia izolację zmian i testowanie poszczególnych fragmentów kodu (Arcelli Fontana i in., 2023; Mumtaz i in., 2021). W specyficznym kontekście mikroserwisów jako dług wskazuje się dodatkowo współdzielone bazy danych (Shared Database) oraz źle zaprojektowane interfejsy API, które wymuszają silne sprzężenie między niezależnymi usługami. Prowadzi to w konsekwencji do efektu kaskadowych zmian oraz nieoczekiwanych awarii w aplikacjach zależnych podczas modyfikacji struktury danych lub kontraktów API (de Toledo i in., 2021).

Metryki ATD

Pomiar i określenie architektonicznego długu technologicznego opiera się na dwóch głównych składowych: kapitale (principal) oraz odsetkach (interest). Kapitał obrazuje szacowany koszt refaktoryzacji, który jest niezbędny do usunięcia samego długu. Przykładowo, jeśli system zawiera ATD w postaci niemodularyzowanego komponentu, to kapitałem jest koszt przebudowy tego komponentu i jego podziału na wiele odrębnych i luźno powiązanych elementów. Odsetki to z kolei wszystkie dodatkowe koszty i wysiłki związane z istnieniem ATD, czyli zasoby, których organizacja nie musiałaby zużywać, gdyby długu w systemie wcześniej nie zaciągnięto. W wadliwie zmodularyzowanym systemie każda drobna zmiana (np. naprawa błędu) może zmuszać deweloperów do modyfikacji wielu innych części oprogramowania, co znacząco wydłuża czas pracy programistów. Zazwyczaj pozycję długu ATD kwalifikuje się do spłaty, jeżeli szacowany koszt kapitału jest mniejszy niż odsetki, które organizacja będzie musiała w przyszłości płacić w związku z dalszym utrzymaniem systemu.

Zagadnienie komplikuje jednak fakt, że niektóre pozycje architektonicznego długu technologicznego bywają wręcz zaraźliwe, przez co płacone przez twórców "odsetki" nie są wartością stałą, lecz zaczynają ulegać kapitalizacji. Aby zdefiniować i wymiernie oszacować płacone odsetki na poziomie struktury systemu, specjaliści monitorują między innymi częstotliwość zmian (ile razy dany artefakt był zmieniany w kolejnych wersjach) oraz

wielkość zmian (liczbę dodanych, usuniętych i zmodyfikowanych linii kodu), które stanowią odzwierciedlenie wysiłku uwarunkowanego pojawieniem się zapachów architektonicznych.

Dodatkowymi wskaźnikami stosowanymi w detekcji miejsc wymagających napraw jest algorytm PageRank oraz miary krytyczności. Wskaźnik ten pomaga precyzyjnie oszacować krytyczność i wagę komponentu dotkniętego defektem na podstawie tego, jak wiele innych części systemu bezpośrednio od niego zależy. W powiązaniu z miarami krytyczności, algorytm PageRank pozwala zidentyfikować pozycje długu mające największy, negatywny wpływ na architekturę, co świadczy o ich centralnej i newralgicznej pozycji w oprogramowaniu. Sama krytyczność długu jest przy tym definiowana specyficznie dla rodzaju wykrytej anomalii. Właściwe miary stosowane w detekcji i ocenie poziomu skomplikowania dla kluczowych zapachów architektonicznych zostały zestawione wcześniej w Tabeli 1.

Możliwym etapem pomiaru jest również analiza strukturalna tzw. Design Structure Matrices (DSM), umożliwiająca ocenę poziomu sprzężenia (coupling) w systemie, z uwagi na fakt, że utrzymanie centralnych, ściśle powiązanych komponentów kosztuje znacznie więcej niż luźno powiązanych elementów peryferyjnych (Fontana i in., 2019; Martini i in., 2018; Martini i in., 2018).

Metryki kodu źródłowego i architektury stanowią fundament ilościowej oceny długu architektonicznego, pozwalając na precyzyjne wskazanie obszarów wymagających interwencji. Kluczową rolę odgrywają metryki sprzężenia, takie jak Fan-In i Fan-Out, które mierzą odpowiednio liczbę przychodzących i wychodzących zależności komponentu, co pozwala na identyfikację modułów o krytycznym znaczeniu dla stabilności systemu. Warto jednak zauważyć, że w ujęciu pragmatycznym naprawa szerokiego sprzężenia (Fan-Out) może wymagać koordynacji wielu zespołów i wielomiesięcznego nakładu pracy. W związku z tym w krótkoterminowej strategii refaktoryzacyjnej wyższy priorytet naprawczy mogą otrzymać wskaźniki o bardziej zlokalizowanym zasięgu operacyjnym, które pojedynczy zespół jest w stanie zniwelować samodzielnie (Yanakiev i in., 2025). Stosowana jest również złożoność cyklomatyczna McCabe’a do oceny skomplikowania logiki wewnątrz jednostek kodu oraz metryki Halsteada do szacowania potencjalnej liczby błędów. Warto jednak zaznaczyć, że wysokie wartości tych metryk uzasadniają refaktoryzację tylko wtedy, gdy dany komponent jest często modyfikowany przez programistów; w przeciwnym razie dług nie generuje bieżących kosztów utrzymania (odsetek), a jego spłata może być nieopłacalna. (Martini i in., 2018).

W literaturze promuje się wykorzystanie macierzy struktury projektowej (Design Structure Matrix – DSM) do obliczania zaawansowanych wskaźników, takich jak koszt propagacji (Propagation Cost), który określa wielkość zmiany w systemie na skutek zmiany w jednym module. Analiza DSM pozwala również na grupowanie komponentów na podstawie ich poziomu sprzężenia, co ułatwia identyfikację silnie powiązanych modułów, które odpowiadają za nieproporcjonalnie dużą część kosztów utrzymania i działań naprawczych w systemie (MacCormack i Sturtevant, 2016). Dodatkowo, analiza częstotliwości zmian poszczególnych artefaktów w połączeniu z rozmiarem tych zmian (mierzonym na podstawie dodanych, usuniętych i zmodyfikowanych linii kodu) pozwala na empiryczne oszacowanie wysiłku konserwacyjnego. Metryki te stanowią praktyczny wskaźnik odsetek od długu architektonicznego, jakie programiści muszą płacić podczas pracy z komponentami obciążonymi zapachami architektonicznymi (Sas i in., 2022).

Prewencja narastania długu

Unikanie wykreowania na etapie projektowym ATD w ogromnej mierze wymaga opracowania efektywnych strategii refaktoryzacji na poziomie struktury i kodu źródłowego. W rozwiązaniach cierpiących na brak odpowiedniej modularności, stosuje się metody automatycznej lub półautomatycznej rekonstrukcji – jedną z powszechnych praktyk jest tu refaktoryzacja oparta na operacji przenoszenia klas obiektowych pomiędzy modułami. (Maikantis i in., 2021). Wstępne badania eksploracyjne potwierdzają, że redukcja problemów strukturalnych poprzez fizyczne usuwanie powszechnych zapachów architektonicznych, takich jak przeciążone klasy (God Class) i zależności cyklicznych znacząco poprawia kluczowe parametry techniczne systemów informatycznych. Eksperymenty wykazały, że refaktoryzacja tych elementów może skrócić czas wykonywania operacji na poziomie metod nawet o 42-47% oraz zmniejszyć zużycie pamięci oprogramowania o 16-20%. Należy jednak pamiętać, że proces ten bywa złożony, a w pojedynczych przypadkach optymalizacja jednego zapachu może nieoczekiwanie pogorszyć wydajność innych, powiązanych ze sobą metod (Arcelli Fontana i in., 2023).

Zalecenia dotyczące ATD koncentrują się na proaktywnym stosowaniu paradygmatów projektowych, które ograniczają ryzyko degradacji architektury systemu. Rekomendowane jest rygorystyczne przestrzeganie zasad SOLID, a w szczególności zasady odwrócenia zależności (Dependency Inversion Principle), która pozwala na budowanie architektury opartej na stabilnych abstrakcjach zamiast na kruchych implementacjach. Aby jednak stosowanie tych paradygmatów było skuteczne w skali całego przedsiębiorstwa, abstrakcyjne reguły projektowe powinny być operacjonalizowane do postaci konkretnych, powtarzalnych

wzorców refaktoryzacyjnych, co z kolei umożliwia zrównoleglenie prac naprawczych pomiędzy wieloma niezależnymi zespołami.

Warto podkreślić, że przy ogromnych przedsięwzięciach i wielomilionowych linijkach kodu konieczne staje się również wdrożenie spójnego systemu detekcji ukierunkowanego na dane. Ponadto, spłata długu architektonicznego i aplikowanie tych wzorców powinny odbywać się w sposób stopniowy i progresywny, co pozwala na systematyczną poprawę jakości kodu bez konieczności całkowitej i ryzykownej przebudowy systemu (Yanakiev i in., 2025). Zaleca się dążenie do wysokiej modułowości, ponieważ liczne zależności między komponentami zwiększają ryzyko powstawania zjawiska „zaraźliwego długu". Polega ono na łańcuchowym rozprzestrzenianiu się suboptymalnych decyzji z jednego modułu na pozostałe (np. poprzez niewłaściwie zaprojektowane interfejsy API lub niefortunne ponowne użycie kodu), co w efekcie prowadzi do niebezpiecznego potęgowania kosztów w postaci odsetek. Aby zapobiec temu zjawisku, inżynierowie powinni proaktywnie monitorować czynniki propagacji długu – takie jak wzrost złożoności systemu czy rosnąca liczba komponentów zależnych – i refaktoryzować kod, zanim kapitał długu drastycznie wzrośnie (Martini i Bosch, 2017).

Dużym jednak wyzwaniem jest priorytetyzacja spłaty długu architektonicznego. Ze względu na trudności w precyzyjnej kwantyfikacji jego wpływu, inżynierowie rzadko dysponują ustrukturyzowaną metodologią i często zmuszeni są polegać na intuicji oraz doświadczeniu. Decyzja o podjęciu działań naprawczych jest zazwyczaj inicjowana, gdy dług osiągnie poziom krytyczny, manifestujący się gwałtownym wzrostem liczby błędów lub incydentów zgłaszanych przez klientów. Co więcej, kluczową rolę w tym procesie odgrywa doświadczenie personelu – to głównie starsi stażem programiści posiadają odpowiednią wiedzę i pewność siebie, aby ATD i priorytetyzować jego refaktoryzację, podczas gdy mniej doświadczeni pracownicy często unikają ingerencji w architekturę systemu (Verdecchia i in., 2021).

W procesie podejmowania decyzji projektowych należy starać się, aby nie dopuścić do zaciągnięcia długu architektonicznego. Bezwzględnie nie wolno dopuszczać do powstawania tzw. nielegalnych zależności (illegal dependencies), które łamią założoną architekturę i wprowadzają szereg problemów, w tym: zwiększone sprzężenie, spadek modułowości, pogorszenie utrzymywalności systemu oraz problemy ze skalowalnością (Yanakiev i in., 2025). Należy unikać świadomego wybierania suboptymalnych rozwiązań projektowych pod presją krótkoterminowych celów, ponieważ prowadzi to do naruszenia zdolności oprogramowania do rozbudowy (Damaceno i in., 2022).

Spłacanie długu

Mitygacja architektonicznego długu technologicznego (ATD) w literaturze przedmiotu opiera się przede wszystkim na procesie refaktoryzacji, która jest uznawana za główną strategię zarządzania i spłacania tego typu zobowiązań (Sousa i in., 2023). Skuteczne wzorce przebudowy obejmują systematyczne stosowanie technik takich jak Move Class, polegającej na przenoszeniu klas w celu optymalizacji spójności i redukcji sprzężeń. W procesie rekonstrukcji architektury technika ta może być automatyzowana przez algorytmy genetyczne. W celu efektywnego i masowego lokalizowania takich możliwości można wykorzystać procesy optymalizacyjne oparte o modele algorytmów genetycznych, które reorganizują strukturę kodu. Aby uniknąć ograniczeń tradycyjnych metod przeszukiwania, algorytmy te oceniają generowane rozwiązania za pomocą funkcji dopasowania, która dąży do maksymalizacji wewnętrznej spójności komponentów przy jednoczesnej minimalizacji niepożądanych sprzężeń zewnętrznych. Zaawansowane podejścia tego typu potrafią dynamicznie modyfikować granice komponentów – poprzez rozbijanie modułów o niskiej spójności oraz scalanie tych wykazujących nadmierne sprzężenie – co pozwala na wygenerowanie nowej, zoptymalizowanej i hierarchicznej struktury całego systemu (Maikantis i in., 2021).

Innym kluczowym podejściem jest modularyzacja, czyli dekompozycja systemu na luźno powiązane komponenty, co zapobiega rozprzestrzenianiu się zmian i pozwala na ewolucję systemu bez generowania nadmiernych kosztów. W praktyce proces ten często polega na hermetyzacji najbardziej złożonych i często modyfikowanych fragmentów kodu wewnątrz funkcjonalności, co zapobiega ich ciągłej ekspozycji na prace programistyczne i redukuje wprowadzanie błędów (Martini i in., 2018).

Aby skutecznie mitygować ATD, literatura zaleca różnorodne strategie zarządzania i refaktoryzacji, które są uzależnione od skali narosłego problemu. Jedną z proaktywnych metod jest budowanie tzw. kredytu technologicznego, co polega na systematycznym inwestowaniu zasobów w poprawę utrzymywalności i ewoluowalności architektury, zanim problemy związane z długiem staną się zauważalne. Badania wskazują jednak, że ze względu na ciągłą presję czasu oraz niepewny, trudny do oszacowania zwrot z takiej inwestycji, strategia ta pozostaje nierzadko jedynie w sferze teoretycznej i rzadko jest w pełni adoptowana w codziennej praktyce zespołów deweloperskich.

W sytuacjach, gdy dług jest już poważny, rekomenduje się podjęcie gruntownej refaktoryzacji (ang. Major Refactoring), polegającej na metodycznym wykorzenianiu problemów architektonicznych. Gruntowna refaktoryzacja to obszerne przedsięwzięcie, które

nierzadko zmusza organizację do poświęcenia dużej ilości zasobów kosztem bieżących działań rozwojowych i wprowadzania nowych funkcjonalności. W najbardziej ekstremalnych przypadkach, gdy architektura staje się całkowicie "skrystalizowana" i blokuje rozwój lub gdy system zachowuje się nieprzewidywalnie, najbardziej opłacalnym wzorcem przebudowy jest napisanie danego komponentu lub całego systemu od nowa. Przepisanie kodu od zera daje unikalną możliwość jednorazowej spłaty całego skumulowanego długu, pozbycia się dawnych, złych praktyk programistycznych oraz modernizacji oprogramowania poprzez wdrożenie nowszych stylów architektonicznych i technologii. Niekiedy stosuje się również strategię świadomego zaniedbania (ang. neglect), która polega na powolniejszym rozwijaniu systemu i akceptacji długu, jeśli koszty jego spłaty przewyższają potencjalne korzyści biznesowe (Verdecchia i in., 2021).

Koszty refaktoryzacji w celu spłaty długu architektonicznego są wysoce uzależnione od rodzaju zapachów architektonicznych, które ten dług tworzą w kodzie. Jak szczegółowo wykazano w Tabeli 1, specjaliści rekomendują priorytetyzację wysiłków refaktoryzacyjnych w taki sposób, aby w pierwszej kolejności pozbywać się długu wynikającego z zależności typu piasta. Charakteryzuje się on najlepszym stosunkiem niskich kosztów naprawy do wysokich korzyści projektowych, podczas gdy restrukturyzacja zależności cyklicznych pozostaje działaniem najbardziej ryzykownym, nierzadko wymagającym setek roboczogodzin. Aby efektywnie kierować procesem spłaty i decydować o kolejności modyfikacji kodu, programiści mogą wspierać się automatycznymi narzędziami, takimi jak Arcan, NDepend czy SonarQube, które identyfikują zapachy architektoniczne w kodzie źródłowym. Co więcej, do oceny hierarchii naprawy błędów architektonicznych z powodzeniem wykorzystuje się opisany wcześniej wskaźnik PageRank. Zgodnie z dobrymi praktykami, proces refaktoryzacji należy rozpoczynać od tych elementów architektury, które wykazują jednocześnie wysoką podatność na błędy oraz posiadają najwyższe wartości tego wskaźnika. Warto jednak zaznaczyć, że nie każdy zidentyfikowany przez automatyczne narzędzia zapach architektoniczny faktycznie stanowi dług wymagający spłaty.

Z badań wynika, że zależności niestabilne są często postrzegane przez programistów jako najmniej przydatne do refaktoryzacji, ponieważ bywają wynikiem celowego i w pełni poprawnego zastosowania sprawdzonych wzorców projektowych. Podobnie, część zdiagnozowanych zależności cyklicznych bywa uznawana za konieczne i naturalne powiązania, na przykład w przypadku wywołań zwrotnych (callbacks) zaimplementowanych w komponentach. Oznacza to, że przed podjęciem decyzji o restrukturyzacji, programiści

muszą ocenić architektoniczny kontekst wykrytego problemu (Fontana i in., 2019; Martini i in., 2018).

Dyskusja

Powszechny w przeanalizowanej literaturze dyskurs nierzadko sprowadza architektoniczny dług technologiczny do problemu stricte matematycznego i strukturalnego, traktując go jako zbiór suboptymalnych decyzji projektowych. Krytyczna analiza zebranych publikacji ukazuje jednak istotną lukę poznawczą: nadmierne skupienie na zautomatyzowanej detekcji (takiej jak algorytmy PageRank czy macierze DSM) marginalizuje socjotechniczny oraz biznesowy kontekst ewolucji oprogramowania. Po pierwsze, traktowanie ATD wyłącznie jako "błędu" inżynierskiego jest podejściem ahistorycznym. Każdy projekt programistyczny funkcjonuje w dynamicznych ramach biznesowych. Decyzje projektowe, które z perspektywy czasu zautomatyzowane narzędzia klasyfikują jako dług architektoniczny, w momencie ich podejmowania nierzadko stanowiły optymalną odpowiedź na ówczesne priorytety. To nie wrodzona wadliwość architektury, lecz naturalna ewolucja i zmiana celów biznesowych sprawiają, że pierwotne ramy ulegają przedawnieniu.

Wymaganie od zespołów deweloperskich nieustannego przebudowywania systemu (np. z wykorzystaniem proponowanych w literaturze algorytmów genetycznych) w odpowiedzi na każdą zmianę jest w realiach rynkowych utopią. Organizacje zmuszone są do stosowania architektonicznych kompromisów (trade-offs), ponieważ koszt osiągnięcia strukturalnego ideału często drastycznie przewyższa potencjalny zwrot z inwestycji. Narzędzia metryczne promują fałszywy obiektywizm, klasyfikują zależności typu piasta jako krytyczny dług architektoniczny, który należy wyeliminować. W praktyce projektowej jednak, centralizacja zależności w jednym module bywa zjawiskiem całkowicie naturalnym i pożądanym. Wynika to z faktu, że określona funkcjonalność lub konkretna reguła biznesowa powinna mieć swoje jedno, wyznaczone miejsce w systemie. Każdy moduł ma określoną odpowiedzialność i tylko on potrafi dane zadanie wykonać poprawnie. Powielanie tej samej logiki biznesowej w wielu różnych komponentach jest błędem. W efekcie, gdy inne części systemu potrzebują dostępu do tej unikalnej logiki, naturalnie odwołują się do jednego, wyspecjalizowanego modułu, tworząc tym samym strukturę piasty.

Ogromną zaletą takiej centralizacji jest łatwość utrzymania logiki biznesowej – gdy zmieniają się wymagania, modyfikacji ulega tylko ten jeden moduł. Literatura wskazuje ten wzorzec jako "zapach architektoniczny" głównie dlatego, że silne sprzężenie wielu komponentów w jedną piastę może rodzić trudności w ich wzajemnej koordynacji oraz

testowaniu. Z tego powodu główną rolą architekta oprogramowania nie jest mechaniczne usuwanie każdego scentralizowanego komponentu, lecz świadome zarządzanie kompromisem. To architekt musi na bazie własnego wyczucia zdecydować, czy w danym przypadku logikę lepiej scentralizować w jednym module (co ułatwia punktowe wprowadzanie zmian, ale tworzy piastę), czy też lepiej rozproszyć lub wręcz powielić pewne mechanizmy w mniejszych częściach (co zmniejsza sprzężenie, ale zwiększa koszty ewentualnej zmiany logiki w przyszłości). Nie istnieją tu jednoznaczne, uniwersalne rekomendacje, które mogłyby zastąpić doświadczenie inżynierskie.

Wreszcie, literatura systematycznie pomija czynnik kognitywny i psychologiczny, skupiając się niemal wyłącznie na refaktoryzacji post factum wspieranej narzędziami statycznymi. Jak zauważono w toku analizy, procesy degeneracji architektury (jej dryf i erozja) rzadko wynikają z celowej ignorancji czy braku wiedzy inżynierskiej. Są one raczej pochodną presji czasu, narzucanej przez paradygmaty ciągłego dostarczania (Continuous Delivery), oraz "zmęczenia materiału", z jakimi na co dzień mierzą się programiści. Skuteczna mitygacja ATD wymaga zatem wyjścia poza zmechanizowaną analizę kodu i wdrożenia na poziomie zarządzania projektem praktyki "strategicznego spowolnienia”. Należy przez to rozumieć świadomą inwestycję w czas polegającą na wbudowaniu w cykl implementacji przestrzeni na pogłębioną refleksję architektoniczną przed ostatecznym zatwierdzeniem zmian, która przeciwstawia się krótkowzrocznej chęci zwiększenia tempa dostarczania funkcjonalności.

Środowisko naukowe, koncentrując się na opracowywaniu coraz doskonalszych algorytmów detekcji, rzadko akcentuje fundamentalną potrzebę budowania dystansu kognitywnego. Dystans ten to celowo zaplanowany moment w procesie deweloperskim, w którym twórca oprogramowania mentalnie odrywa się od zaimplementowanej, doraźnej funkcjonalności i mikroskopijnego poziomu kodu. Celem tego zabiegu nie jest poszukiwanie standardowych błędów implementacyjnych (co z powodzeniem realizują zautomatyzowane lintery czy testy), lecz ponowna, z krytycznej perspektywy, konfrontacja nowo powstałego rozwiązania z pierwotnymi, makroskopowymi założeniami całego modułu. Brak takiego etapu ludzkiej walidacji – swoistego "kroku w tył" – sprawia, że seria drobnych architektonicznych kompromisów, z których każdy z osobna wydaje się niegroźny i poprawny dla danego zadania, ostatecznie akumuluje się w postaci potężnego długu. Zaufanie wyłącznie narzędziom ciągłej integracji, bez zapewnienia inżynierom czasu na weryfikację założeń i wyjście poza schemat bieżącego zadania, sprawia, że zespoły tracą z oczu ewolucję całego systemu, nieświadomie potęgując jego degradację.

Podsumowanie

Niniejsza praca stanowi ustrukturyzowany przegląd literatury przedmiotu w zakresie zjawiska architektonicznego długu technologicznego (ATD), oparty na analizie 15 publikacji z bazy Scopus. Celem artykułu było usystematyzowanie definicji długu architektonicznego, identyfikacja metod jego wykrywania oraz ewaluacja strategii jego spłaty w świetle wyzwań biznesowych i inżynierskich. Przeprowadzone analizy potwierdzają, że ATD jest zjawiskiem narastającym wraz z ewolucją systemów informatycznych, a jego ignorowanie prowadzi do degradacji struktury oprogramowania i drastycznego wzrostu kosztów utrzymania, które mogą pochłonąć znaczną część budżetu projektu.

Odpowiadając na pierwsze pytanie badawcze (RQ1), w literaturze architektoniczny dług technologiczny definiuje się jako konsekwencję suboptymalnych decyzji dotyczących struktury systemu, wyboru technologii lub procesów rozwoju. W przeciwieństwie do długu na poziomie kodu, ATD dotyczy „dużych" decyzji projektowych, takich jak podział na podsystemy czy dobór frameworków, i wpływa głównie na wewnętrzne cechy systemu: jego łatwość utrzymania oraz zdolność do dalszej ewolucji.

W kontekście detekcji długu (RQ2), badanie wykazało, że najskuteczniejszymi narzędziami są zapachy architektoniczne (Architectural Smells) oraz metryki sprzężenia. Do najczęściej wskazywanych anomalii należą zależności cykliczne, zależności typu piasta oraz niestabilne zależności. Ilościowa ocena długu opiera się na monitorowaniu częstotliwości i rozmiaru zmian w kodzie, a także na wykorzystaniu zaawansowanych wskaźników, takich jak algorytm PageRank czy macierze DSM, które pozwalają oszacować koszt propagacji zmian w systemie.

Odpowiedź na trzecie pytanie badawcze (RQ3) wskazuje, że główną metodą spłacania ATD jest refaktoryzacja oparta na wzorcach projektowych i zasadach SOLID, ze szczególnym uwzględnieniem zasady odwrócenia zależności. Skuteczne techniki obejmują operację przenoszenia klas, często automatyzowaną za pomocą algorytmów genetycznych oraz głęboką dekompozycję systemów na luźno powiązane komponenty. Przeprowadzona dyskusja zwraca uwagę, że wdrażanie tych rozwiązań wymaga nieustannych kompromisów (trade-offs), a ślepe podążanie za metrykami kodu może być błędem. Zaznaczono na przykład, że powszechnie piętnowane w literaturze scentralizowane zależności typu piasta mogą w praktyce projektowej stanowić pożądany wzorzec architektoniczny chroniący przed powielaniem logiki biznesowej.

Ostatecznym wnioskiem płynącym z artykułu jest to, że optymalne zarządzanie ATD nie może opierać się wyłącznie na zmechanizowanej analizie kodu i narzędziach

automatycznych. Skuteczna prewencja i mitygacja długu wymagają uwzględnienia czynników kognitywnych i organizacyjnych – w szczególności wdrożenia na poziomie zarządzania projektem praktyk „strategicznego spowolnienia" oraz budowania przez programistów „dystansu kognitywnego". To właśnie doświadczenie inżynierskie i czas na refleksję pozwalają świadomie balansować pomiędzy presją ciągłego dostarczania a utrzymaniem jakości architektury. Przyszłe kierunki badań powinny koncentrować się na opracowaniu bardziej precyzyjnych metod kwantyfikacji wpływu długu na cele biznesowe oraz na badaniu psychologicznych i organizacyjnych aspektów pracy zespołów, co ułatwi inżynierom podejmowanie decyzji o momentach inicjowania prac refaktoryzacyjnych.

Bibliografia:

Arcelli Fontana, F., Camilli, M., Rendina, D., Taraboi, A. G., Trubiani, C. (2023). Impact of Architectural Smells on Software Performance: An Exploratory Study. *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering, EASE '23*, 22-31. https://doi.org/10.1145/3593434.3593442

Bochicchio, M., Sas, D., Gilardi, A., Fontana, F. A. (2025). An empirical study on architectural smells through a pipeline for continuous technical debt assessment. *Information and Software Technology*, *185*(C), 107783. https://doi.org/10.1016/j.infsof.2025.107783

Damaceno, K., Nakagawa, E., Braga, R. (2022). Towards an Understanding of Technical Debt in Reference Architectures. *Proceedings of the XXXVI Brazilian Symposium on Software Engineering, SBES '22*, 257-262. https://doi.org/10.1145/3555228.3555270

Fontana, F. A., Pigazzini, I., Raibulet, C., Basciano, S., Roveda, R. (2019). PageRank and criticality of architectural smells. *Proceedings of the 13th European Conference on Software Architecture - 2, ECSA '19*, 197-204. https://doi.org/10.1145/3344948.3344982

MacCormack, A., Sturtevant, D. J. (2016). Technical debt and system architecture: The impact of coupling on defect-related activity. *Journal of Systems and Software*, *120*(C), 170-182. https://doi.org/10.1016/j.jss.2016.06.007

Maikantis, T., Tsintzira, A.-A., Ampatzoglou, A., Arvanitou, E.-M., Chatzigeorgiou, A., Stamelos, I., Bibi, S., Deligiannis, I. (2021). Software Architecture Reconstruction via a Genetic Algorithm: Applying the Move Class Refactoring. *Proceedings of the 24th Pan-Hellenic Conference on Informatics, PCI '20*, 135-139. https://doi.org/10.1145/3437120.3437292

Martini, A., Bosch, J. (2017). On the interest of architectural technical debt: Uncovering the contagious debt phenomenon. *Journal of Software: Evolution and Process*, *29*(10). https://doi.org/10.1002/smr.1877

Martini, A., Fontana, F. A., Biaggi, A., Roveda, R. (2018). Identifying and prioritizing architectural debt through architectural smells: A case study in a large software company. *Lecture Notes in Computer Science*, *11048 LNCS*, 320-335. https://doi.org/10.1007/978-3-030-00761-4_21

Martini, A., Sikander, E., Madlani, N. (2018). A semi-automated framework for the identification and estimation of Architectural Technical Debt: A comparative case-study on the modularization

of a software component. *Information and Software Technology*, *93*(C), 264-279. https://doi.org/10.1016/j.infsof.2017.08.005

Mumtaz, H., Singh, P., Blincoe, K. (2021). A systematic mapping study on architectural smells detection. *Journal of Systems and Software*, *173*, 110885. https://doi.org/10.1016/j.jss.2020.110885

Sas, D., Avgeriou, P., Pigazzini, I., Arcelli Fontana, F. (2022). On the relation between architectural smells and source code changes. *Journal of Software: Evolution and Process*, *34*(1), e2398. https://doi.org/10.1002/smr.2398

Sousa, A., Rocha, L., Britto, R. (2023). Architectural Technical Debt - A Systematic Mapping Study. *Proceedings of the XXXVII Brazilian Symposium on Software Engineering*, 196-205. https://doi.org/10.1145/3613372.3613399

de Toledo, S. S., Martini, A., Sjøberg, D. I. K. (2021). Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study. *Journal of Systems and Software*, *177*, 110968. https://doi.org/10.1016/j.jss.2021.110968

Verdecchia, R., Kruchten, P., Lago, P., Malavolta, I. (2021). Building and evaluating a theory of architectural technical debt in software-intensive systems. *Journal of Systems and Software*, *176*, 110925. https://doi.org/10.1016/j.jss.2021.110925

Yanakiev, I., Lazar, B.-M., Capiluppi, A. (2025). Applying SOLID principles for the refactoring of legacy code: An experience report. *Journal of Systems and Software*, *220*, 112254. https://doi.org/10.1016/j.jss.2024.112254

Nota o autorze

mgr inż. Jarosław Bochenek, zatrudniony w: Wyższa Szkoła Biznesu – National Louis University w Nowym Sączu. Zainteresowania badawcze: optymalizacja procesów na uczelniach z wykorzystaniem AI (BPM) oraz inżynieria oprogramowania.