Załącznik nr 2: do Wniosku o przeprowadzenie postępowania w sprawie nadania stopnia doktora habilitowanego.

Autoreferat

Tytuł osiągnięcia:

„Model 1+5 widoków architektonicznych współpracujących systemów informatycznych"

dr inż. Tomasz Górski
Instytut Informatyki,
Wydział Matematyki, Fizyki i Informatyki
Uniwersytet Gdański

Gdańsk, kwiecień 2024

I. Imię i nazwisko

Tomasz Górski

II. Posiadane dyplomy, stopnie naukowe lub artystyczne

Tabela 1. Dane uzyskanego stopnia doktora nauk technicznych.

Uzyskany stopień naukowy:	**Doktor nauk technicznych**
Dyscyplina:	Informatyka
Specjalność:	Systemy informatyczne
Nadany uchwałą:	Rada Wydziału Cybernetyki, Wojskowej Akademii Technicznej
Data nadania:	14 listopada 2000 r.
Tytuł rozprawy:	Symulacyjna metoda badania wpływu niejednorodności obciążenia i decentralizacji rozpraszania zadań na efektywność funkcjonowania rozproszonych sieci komputerowych
Promotor:	płk dr hab. inż. Tadeusz Nowicki
Recenzenci:	dr hab. inż. Eugeniusz Toczyłowski płk dr hab. inż. Andrzej Chojnacki
Nagrody:	W dniu 15 listopada 2001 roku praca doktorska otrzymała nagrodę II stopnia w Wojskowej Akademii Technicznej w konkursie na najlepszą pracę doktorską roku akademickiego 2000/2001.

Tabela 2. Dane uzyskanego tytułu zawodowego magistra inżyniera.

Uzyskany tytuł:	**Magister inżynier**
Kierunek:	Informatyka
Zakres:	Systemy informatyczne
Nadany przez:	Wydział Cybernetyki, Wojskowej Akademii Technicznej
Data nadania:	12 lipca 1997 r.
Tytuł rozprawy:	Symulacyjna metoda badania efektywności rozproszonych sieci komputerowych
Promotor:	płk dr hab. inż. Tadeusz Nowicki

III. Informacja o dotychczasowym zatrudnieniu w jednostkach naukowych lub artystycznych

Tabela 3. Okresy zatrudnienia i stanowiska w jednostkach naukowych.

Okres zatrudnienia	Stanowisko	Jednostka naukowa
01.10.2022 – obecnie	Adiunkt badawczo-dydaktyczny	Instytut Informatyki, Wydział Matematyki, Fizyki i Informatyki, Uniwersytet Gdański
01.03.2017 – 30.09.2022	Adiunkt badawczo-dydaktyczny	Katedra Informatyki, Wydział Mechaniczny, Akademia Marynarki Wojennej
01.11.2005 – 31.03.2018	Adiunkt naukowo-dydaktyczny	Instytut Systemów Informatycznych, Wydział Cybernetyki, Wojskowa Akademia Techniczna

IV. Omówienie osiągnięć, o których mowa w art. 219 ust. 1 pkt. 2 ustawy z dnia 20 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2021 r. poz. 478 z późn. zm.).

IV.1. Osiągnięcie naukowe

Osiągnięciem naukowym jest jednotematyczny cykl publikacji naukowych, pt.

Model 1+5 widoków architektonicznych współpracujących systemów informatycznych

IV.2. Cykl publikacji

Przedstawiony do oceny cykl publikacji składa się z dziewięciu artykułów z listy JCR:

[A1] Górski, T. SmarTS: A Java package for smart contract test suite generation and execution. SoftwareX, **2024**, 26, 101698. https://doi.org/10.1016/j.softx.2024.101698

[A2] Górski, T. Smart Contract Design Pattern for Processing Logically Coherent Transaction Types. Applied Sciences, **2024**, 14(6), 2224. https://doi.org/10.3390/app14062224

[A3] Górski, T., Integration Flows Modeling in the Context of Architectural Views, IEEE Access, **2023**, 11, 35220–35231. https://doi.org/10.1109/ACCESS.2023.3265210

[A4] Górski, T. UML Profile for Messaging Patterns in Service-Oriented Architecture, Microservices, and Internet of Things. Applied Sciences, **2022**, 12(24), 12790. https://doi.org/10.3390/app122412790

[A5] Górski, T., The k + 1 Symmetric Test Pattern for Smart Contracts, Symmetry, **2022**, 14(8) 1686. https://doi.org/10.3390/sym14081686

[A6] Górski, T., Reconfigurable Smart Contracts for Renewable Energy Exchange with Re-Use of Verification Rules, Applied Sciences, **2022**, 12(11) 5339. https://doi.org/10.3390/app12115339

[A7] Górski, T., The 1+5 Architectural Views Model in Designing Blockchain and IT System Integration Solutions, Symmetry, **2021**, 13(11), 2000. https://doi.org/10.3390/sym13112000

[A8] Górski, T.; Woźniak, P. A., Optimization of business process execution in services architecture: a systematic literature review, IEEE Access, **2021**, 9, 111833–111852. https://doi.org/10.1109/ACCESS.2021.3102668

[A9] Górski, T.; Bednarski, J., Applying Model-Driven Engineering to Distributed Ledger Deployment, IEEE Access, **2020**, 8, 118245–118261. https://doi.org/10.1109/ACCESS.2020.3005519

Tabela 4 zawiera podsumowanie danych dotyczących publikacji wchodzących w skład osiągnięcia naukowego.

Tabela 4. Podsumowanie danych publikacji.

Publikacja	**Udział habilitanta**	**Rok publikacji**	**Impact Factor (wg roku publikacji)**	**Liczba punktów MNiSW**
[A1]	100%	2024	3,4	200
[A2]	100%	2024	2,7	100
[A3]	100%	2023	3,9	100
[A4]	100%	2022	2,7	100
[A5]	100%	2022	2,7	70
[A6]	100%	2022	2,7	100
[A7]	100%	2021	2,940	70
[A8]	70%	2021	3,476	100
[A9]	70%	2020	3,367	100
SUMA			**27,883**	**940**

Dla lat 2023-2024 został podany aktualny Impact Factor czasopism zgodnie z 2022 Journal Citation Reports.

Oświadczenia współautorów publikacji [A8] oraz [A9] dotyczące ich wkładu w przygotowaniu tych publikacji załączono do wniosku.

IV.3. Podsumowanie nowego wkładu publikacji

Przedstawiony do oceny cykl publikacji wnosi do dyscypliny „Informatyka techniczna i telekomunikacja" następujące oryginalne osiągnięcia:

- Model widoków architektonicznych 1+5 dający możliwość opisu współpracujących systemów informatycznych realizujących wspólne procesy biznesowe [A7]. Model ten zawiera sześć widoków architektonicznych: *Integrowanych procesów* (ang. Integrated processes), *Przypadków użycia* (ang. Use cases), *Logiczny* (ang. Logical), *Kontraktów* (ang. Contracts), *Integrowanych usług* (ang. Integrated services) oraz *Wdrożeniowy* (ang. Deployment). Trzy spośród tych widoków to nowe, autorskie propozycje habilitanta. Są to widoki: *Integrowanych procesów*, *Kontraktów* oraz *Integrowanych usług*. Model 1+5 uwzględnia projektowanie współpracujących systemów informatycznych. W szczególności widok *Integrowanych usług* dostarcza metod oraz narzędzi modelowania przepływów

integracyjnych. Model dostosowany jest także do projektowania rozwiązań w technologii łańcucha bloków (ang. blockchain). W tym kontekście widok *Kontraktów* jest predystynowany do projektowania inteligentnych kontraktów (ang. smart contracts) technologii łańcucha bloków.

- Diagram przepływów integracyjnych (ang. Integration flow diagram) będący rozszerzeniem diagramu aktywności UML [A7].
- Metody modelowania przepływów integracyjnych: usługowego oraz biznesowego [A3]. Zaproponowano definicje przepływów: usługowego oraz biznesowego.
- Profil języka Unified Modeling Language (UML), *UML Profile for Messaging Patterns* zawierający konstrukcje semantyczne do modelowania przepływów integracyjnych [A4].
- Metoda opisu architektury rozwiązania uruchamiania procesów biznesowych z dynamiczną alokacją usług [A8]. Metoda obejmuje widoki: *Integrowanych procesów*, *Integrowanych usług* oraz *Wdrożeniowy*.
- Wzorzec projektowania inteligentnych kontraktów, technologii łańcucha bloków AdapT v2.0 umożliwiający przetwarzanie spójnych logicznie typów transakcji [A2].
- Implementacja warstwy abstrakcyjnej wzorca AdapT v2.0 w języku Java, niezależna od konkretnego inteligentnego kontraktu [A2].
- Wzorzec *Smart Contract Design Pattern* (SCDP) projektowania i wdrażania inteligentnych kontraktów z jawną deklaracją reguł weryfikacji [A6]. Zaproponowano definicje: *reguły weryfikacji*, *konfiguracji inteligentnego kontraktu* oraz *wyrażenia ewaluacji*. Wzorzec umożliwia rekonfigurację listy reguł weryfikacji w czasie wykonywania.
- Profil języka UML, *UML Profile for Smart Contracts* zawierający konstrukcje semantyczne do modelowania inteligentnych kontraktów [A6].
- Metoda redukcji zestawu testów k+1 dla inteligentnego kontraktu [A5].
- Pakiet oprogramowania SmarTS implementujący metodę redukcji zestawu testów k+1 dla inteligentnego kontraktu [A1].
- Metoda modelowania widoku *Wdrożeniowego* dla technologii łańcucha bloków [A9].
- Profil języka UML, *UML Profile for Distributed Ledger Deployment* obejmujący stereotypy i wartości oznaczone dla środowiska łańcucha bloków R3 Corda w widoku *Wdrożeniowym* [A9].
- Transformacja *UML2Deployment* z inżynierii sterowanej modelami (ang. Model-Driven Engineering, MDE) generująca skrypty wdrożeniowe dla środowiska łańcucha bloków R3 z modelu w widoku *Wdrożeniowym* [A9].

Poniżej przedstawiono podsumowanie nowego wkładu w poszczególnych publikacjach wchodzących w skład osiągnięcia naukowego:

[A1] – Praca własna koncentrująca się na widoku architektonicznym *Kontraktów* modelu 1+5. Wkładem jest opracowany nowy pakiet oprogramowania SmarTS. Pakiet implementuje metodę redukcji zestawu testów k+1 dla inteligentnych kontraktów technologii łańcucha bloków. Metoda k+1 zapewnia pełne pokrycie na poziomie metod, linii, instrukcji oraz gałęzi kodu źródłowego. Opracowane oprogramowanie generuje i uruchamia minimalny zestaw przypadków testowych. Warstwowa konstrukcja pakietu zapewnia możliwość testowania inteligentnych kontraktów z dowolnej dziedziny przedmiotowej. Warunkiem jest, aby kontrakt zaprojektowany był zgodnie z określonym wzorcem projektowym. Testy wydajności pokazują, że czasy wykonania zestawu testów dla inteligentnych kontraktów z liczbą reguł weryfikacji z przedziału <1; 8> mieszczą się w przedziale czasowym <0,0016; 0,0082> milisekund. Dzięki temu metoda i opracowany pakiet mogą być z powodzeniem stosowane w ciągłej integracji i dostarczaniu oprogramowania.

[A2] – Praca własna koncentrująca się na widoku architektonicznym *Kontraktów* modelu 1+5. Wkładem jest wzorzec projektowania inteligentnych kontraktów, technologii łańcucha bloków AdapT v2.0 umożliwiający przetwarzanie dowolnej liczby spójnych logicznie typów transakcji. W aktualnej wersji wzorca przeprojektowano jego warstwę abstrakcyjną wykorzystując mechanizmy programowania obiektowego i funkcyjnego. Implementacja warstwy abstrakcyjnej w języku Java jest niezależna od konkretnego inteligentnego kontraktu. Implementację warstwy konkretnej wzorca napisano w języku Java na przykładzie inteligentnego kontraktu dla przesyłania energii w sieciach prosumentów. Praca zawiera metryki ponownego użycia kodu źródłowego oraz redundancji reguł weryfikacji. Dokonano analizy wartości zaproponowanych metryk dla wzorca AdapT v2.0. Wartość redundancji reguł weryfikacji w inteligentnym kontrakcie zaprojektowanym zgodnie ze wzorcem wynosi 0%. Praca zawiera również wyniki testów wydajności inteligentnego kontraktu zaprojektowanego zgodnie ze wzorcem AdapT v2.0 dla liczby transakcji w zakresie od 100 000 do 10 000 000. Czas ewaluacji 10 000 000 transakcji przez inteligentny kontrakt zaprojektowany zgodnie ze wzorcem AdapT v2.0 wynosi poniżej 0,25 sekundy. Habilitant udostępnił kod źródłowy wzorca AdapT v2.0 w publicznym repozytorium [1].

[A3] – Praca własna koncentrująca się na następujących widokach architektonicznych modelu 1+5: *Integrowanych procesów*, *Przypadków użycia* oraz *Integrowanych usług*. Przedstawienie dwóch metod modelowania przepływów komunikatów na poziomie usługowym oraz procesu biznesowego. Przepływ usługowy pozwala na modelowanie komunikacji pomiędzy

systemami/usługami na poziomie wysyłania komunikatów. Natomiast przepływ biznesowy modeluje cały przebieg interakcji pomiędzy współpracującymi aplikacjami. Metody wykorzystują *diagram przepływów integracji* (ang. Integration flow diagram), który jest wyspecjalizowaną wersją diagramu aktywności UML. W artykule wykorzystano autorski widok architektoniczny *Integrowanych procesów*. Zadanie procesu biznesowego z tego widoku definiuje kontekst przepływu biznesowego. Co więcej, widok *Przypadków użycia* identyfikuje zakres integracji w ramach wymagań. Wszystkie te widoki należą do modelu widoków architektonicznych 1+5. W artykule przedstawiono rozszerzoną wersję profilu UML, *UML Profile for Messaging Patterns* zawierającego wzorce przesyłania komunikatów. Profil ten został wzbogacony o stereotypy dotyczące wzorców przesyłania wiadomości dla platformy Apache Camel, interesariuszy zaangażowanych w komunikację oraz aktualne stereotypy dotyczące komponentów strukturalnych. Metody zostały wykorzystane w projekcie integracji przesyłania zleceń i potwierdzeń pomiędzy aplikacjami biznesowymi domu maklerskiego i giełdy papierów wartościowych.

[A4] – Praca własna koncentrująca się na widoku architektonicznym *Integrowanych usług* modelu 1+5. W artykule przedstawiono zestaw wzorców przesyłania komunikatów dla architektury zorientowanej na usługi, mikrousługi (ang. microservices) i protokołów przesyłania wiadomości dla Internetu Rzeczy (ang. Internet-of-Thing, IoT). W artykule opisano wybrane wzorce będące efektem bieżących prac badawczych, np. [2], [3]. Uwzględniono również wzorce wprowadzone w środowiskach z otwartym kodem (ang. open source), takich jak ZeroMQ. W zestawie znajdują się także Wzorce Integracji Przedsiębiorstw (ang. Enterprise Integration Patterns) [6]. Wszystkie rozważane wzorce przesyłania wiadomości zostały opisane przy użyciu mechanizmu rozszerzalności UML jakim są stereotypy. Ich pełny zestaw został uwzględniony w profilu *UML Profile for Messaging Patterns* dla wzorców przesyłania komunikatów. Artykuł przedstawia także sposób modelowania przepływu integracji. W przykładach ilustracyjnych zarówno *diagram przepływu integracji*, jak i profil zostały wykorzystane do opisu komunikacji w kontekście widoku *Integrowanych usług* modelu widoków architektonicznych 1+5. Profil został zaprojektowany w narzędziu Visual Paradigm. Habilitant udostępnił profil w publicznym repozytorium [8].

[A5] – Praca własna koncentrująca się na widoku architektonicznym *Kontraktów* modelu 1+5. Inteligentny kontrakt weryfikuje spełnienie warunków umożliwiających wykonanie transakcji pomiędzy węzłami sieci łańcucha bloków. Te sterowane programowo warunki logiczne nazywano regułami weryfikacji. Wraz ze wzrostem liczby warunków szybko rośnie złożoność testowania inteligentnych kontraktów. Celem artykułu było zaproponowanie wzorca testowania

inteligentnego kontraktu k+1, który znacząco ogranicza wymaganą liczbę przypadków testowych. Dla wyrażenia ewaluacyjnego z siedmioma regułami weryfikacji zastosowanie wzorca k+1 zmniejsza liczbę przypadków testowych o ponad 90% w stosunku do pełnego pokrycia kombinacji wartości logicznych. Wraz ze wzrostem liczby warunków logicznych poziom redukcji wzrasta. W rezultacie koszty przygotowania i utrzymania zestawów przypadków testowych mogą zostać znacznie obniżone. Należy podkreślić, że czas wykonania testu można skrócić nawet o 3 rzędy wielkości (z sekund do milisekund). Tego typu podejście może przynieść wymierne korzyści w przypadku testów regresyjnych, zwłaszcza gdy jest stosowane w ciągłej integracji, dostarczaniu i wdrażaniu oprogramowania. Habilitant udostępnił kod źródłowy zaimplementowanego wzorca k+1 w publicznym repozytorium [9].

[A6] – Praca własna koncentrująca się na widoku architektonicznym *Kontraktów* modelu 1+5. W artykule przedstawiono wzorzec *Smart Contract Design Pattern* projektowania i wdrażania inteligentnych kontraktów z jawną deklaracją zasad weryfikacji. Habilitant zaproponował definicje: *reguły weryfikacji, konfiguracji inteligentnego kontraktu* oraz *wyrażenia ewaluacji*. Wzorzec zapewnia możliwość rekonfiguracji listy reguł weryfikacji inteligentnych kontraktów w czasie wykonywania w celu dostosowania do różnych typów transakcji. Zaproponowane podejście do projektowania pozwala także na ponowne wykorzystanie reguł weryfikacji pomiędzy różnymi konfiguracjami inteligentnego kontraktu oraz pomiędzy różnymi inteligentnymi kontraktami. W artykule przedstawiono stereotypy z zaproponowanego dedykowanego profilu UML do projektowania inteligentnych kontraktów, *UML Profile for Smart Contracts*. Profil został zaprojektowany w narzędziu Visual Paradigm. Habilitant udostępnił profil w publicznym repozytorium [10]. Implementacja wzorca została wykonana w obiektowym języku Java. Reguły weryfikacji konstruowane są w postaci klas. Implementacja wykorzystuje polimorfizm i kontroluje dziedziczenie, używając klas zapieczętowanych (ang. sealed) z pozwoleniem na specjalizację tylko dla wybranych klas ostatecznych (ang. final). Przedstawiony mechanizm zapewnia dwie pożądane właściwości w projektowaniu inteligentnych kontraktów: ponowne wykorzystanie i bezpieczeństwo. Zastosowanie wzorca ilustruje przykład wymiany energii odnawialnej w społeczności prosumentów i pomiędzy różnymi społecznościami.

[A7] – W pracy własnej habilitant przedstawił autorski model widoków architektonicznych 1+5 do projektowania współpracujących systemów informatycznych. Model zawiera sześć widoków architektonicznych: *Integrowanych procesów, Przypadków użycia, Logiczny, Kontraktów, Integrowanych usług* oraz *Wdrożeniowy*. Trzy spośród tych widoków to autorskie propozycje habilitanta. Są to widoki: *Integrowanych procesów, Kontraktów* oraz

Integrowanych usług. Widok *Integrowanych procesów* został dołączony do opisu architektonicznego systemów informatycznych i zawiera modele procesów biznesowych. Aspekty integracji obejmują dwa pozostałe nowe widoki architektoniczne: *Kontraktów* oraz *Integrowanych usług*. Habilitant wprowadził dwa nowe profile języka UML: *UML for Integration Flows* [15] oraz *UML Profile for Distributed Ledger Deployment* [11]. W artykule przedstawiono *diagram przepływów integracji* będący rozwinięciem diagramu aktywności języka UML. Dla projektowania rozwiązań w technologii łańcucha bloków habilitant zaproponował wzorzec projektowania inteligentnych kontraktów *Smart Contract Design Pattern*. Wykonano implementację tego wzorca w środowisku łańcucha bloków R3 Corda. Kod źródłowy udostępniono w publicznym repozytorium [16]. Opisano trzy studia przypadków, w których wykorzystano model 1+5 do projektowania rozwiązań integracyjnych. Model jest przydatny do projektowania zarówno scentralizowanych środowisk integracyjnych z magistralą usług korporacyjnych (ang. Enterprise Service Bus), jak i rozproszonych rozwiązań łańcucha bloków z połączeniami typu punkt-punkt (ang. point-to-point).

[A8] – Praca współautorska koncentrująca się na następujących widokach architektonicznych modelu 1+5: *Integrowanych procesów*, *Integrowanych usług* oraz *Wdrożeniowym*. W artykule habilitant przedstawił sposób opisu architektury takiego typu rozwiązania obejmującą proces biznesowy (widok *Integrowanych procesów*), identyfikację usług (widok *Integrowanych usług*) oraz przypisanie usług do fizycznych komponentów realizujących te usługi (widok *Wdrożeniowy*). Artykuł stanowi systematyczny przegląd literatury w obszarze dostępnych metod optymalizacji wykonania procesów biznesowych. Metody przedstawione w analizowanych artykułach podzielono na trzy etapy optymalizacji procesów biznesowych: alokacja zasobów, kompozycja usług i planowanie usług. Najliczniejszą grupę badaczy przyciąga etap kompozycji usług, w której znajduje się zdecydowana większość ze 119 artykułów objętych przeglądem. Najpopularniejsze są algorytmy genetyczne. Stosowane są głównie metody heurystyczne, optymalizujące procesy biznesowe w czasie ich wykonywania.

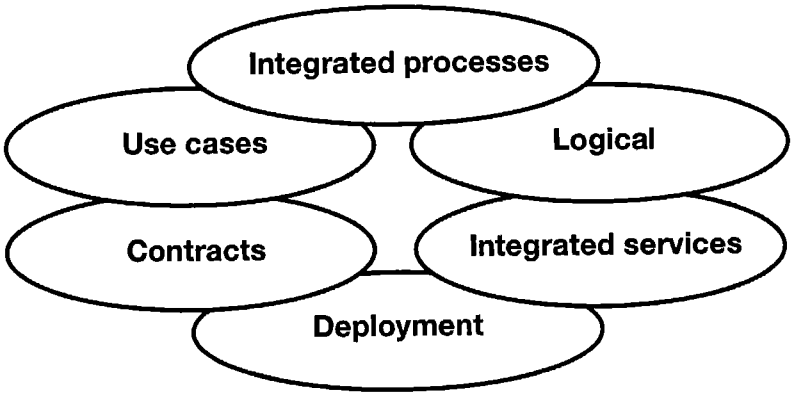
[A9] – Praca współautorska koncentrująca się na następujących widokach architektonicznych modelu 1+5: *Kontraktów* oraz *Wdrożeniowym*. Artykuł zawiera sposób modelowania widoku *Wdrożeniowego* dla rozwiązań w technologii łańcucha bloków. Na poziomie wdrożeniowym wpływ na modelowanie tego typu rozwiązań ma wybrana platforma łańcucha bloków. Przedstawiono rozwiązanie dla środowiska R3 Corda. Zaproponowano nowe stereotypy i wartości oznaczone (ang. tagged values) języka UML na potrzeby modelowania poziomu wdrożeniowego rozwiązań łańcucha bloków. Utworzono nowy profil języka UML obejmujący stereotypy i wartości oznaczone dla R3 Corda w wersji 4.3. Nowe stereotypy zaproponowano

dla węzłów, usług oraz protokołów komunikacyjnych. Natomiast wartości oznaczone zdefiniowano dla parametrów wdrożeniowych węzła sieci łańcucha bloków. Profil nazwano *UML Profile for Distributed Ledger Deployment* i udostępniono w publicznym repozytorium [11]. Ponadto, zapewniono spójność pomiędzy modelami i skryptami wdrażania konfiguracji. Zastosowano inżynierię sterowaną modelami. Zaprojektowano transformację modelu w skrypt wdrożeniowy *UML2Deployment* dla środowiska R3 Corda w wersji 4.3. Model wdrożeniowy UML jest źródłem, podczas gdy skrypt wdrażania Gradle Groovy jest celem transformacji. Dostarczono kompletne rozwiązanie, włączając transformację do narzędzia modelowania Visual Paradigm. Zaprojektowano także dedykowaną wtyczkę umożliwiającą walidację wygenerowanych skryptów wdrożeniowych.

IV.4. Szczegółowe omówienie osiągnięcia naukowego

Wytwarzanie złożonych systemów informatycznych wymusza konieczność opisu ich architektury. Norma ISO/IEC/IEEE 42010:2011 wprowadza terminy definiujące opis architektury inżynierii oprogramowania [17]. Opis architektury to produkt pracy używany do wyrażenia architektury. Architektura obejmuje kluczowe idee lub cechy systemu zawarte w jego częściach, relacje oraz zasady jego projektowania i rozwoju. Widok architektury pokazuje architekturę systemu z określonej perspektywy. Opis architektury obejmuje widoki i modele. Kruchten [18] przedstawił model architektury oprogramowania 4+1, który obejmuje pięć widoków architektonicznych: Scenariuszy, Logiczny, Wytwarzania, Procesów i Fizyczny. Nazwy scenariusz i przypadek użycia używane są w tym modelu zamiennie. Brakuje w tym modelu możliwości opisu procesów biznesowych oraz uwzględnienia wymagań poza funkcjonalnych. Model ten jest przeznaczony do projektowania pojedynczego systemu informatycznego. Istotą opracowanego modelu widoków architektonicznych 1+5 jest możliwość opisu współpracujących systemów informatycznych realizujących wspólne procesy biznesowe i wymieniających informacje [A7]. Model ten zawiera sześć widoków architektonicznych: *Integrowanych procesów*, *Przypadków użycia*, *Logiczny*, *Kontraktów*, *Integrowanych usług* oraz *Wdrożeniowy*. Trzy spośród tych widoków to nowe, autorskie propozycje habilitanta: *Integrowanych procesów*, *Kontraktów* oraz *Integrowanych usług*.

Rysunek 1 przedstawia model widoków architektonicznych 1+5.



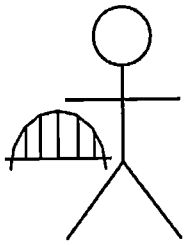
Rysunek 1. Model widoków architektonicznych 1+5.

W widoku *Integrowanych procesów* możemy modelować procesy biznesowe. Po drugie, możemy zidentyfikować wymagane wsparcie przez systemy oprogramowania. Ponadto możemy przyjrzeć się wymaganej współpracy między tymi systemami oprogramowania. Generalnie, w procesie biznesowym mamy zadania ludzkie i zautomatyzowane. Te ostatnie stają się usługami. Automatyzują realizację procesu biznesowego. Możemy modelować procesy na diagramie aktywności UML lub diagramie procesu biznesowego w notacji Business

Process Model and Notation (BPMN). W widoku *Przypadków użycia* możemy opisać funkcje systemu oprogramowania. Zakres funkcjonalności przedstawiony jest na diagramie przypadków użycia UML. W widoku *Logicznym* możemy zaprojektować realizacje przypadków użycia zidentyfikowanych w widoku *Przypadków użycia*. W tym celu zwykle używane są trzy typy diagramów UML: klas, sekwencji i komunikacji. Diagram klas UML można wykorzystać także do przedstawienia struktury encji biznesowych wykorzystywanych w procesach biznesowych. Widok *Kontraktów* pokazuje warunki umów nałożonych na współpracujące strony. W tym widoku wykorzystuje się stereotypy z języka modelowania architektury zorientowanej na usługi (SoaML) do określenia dostawców i konsumentów usług. Stereotyp <<Provider>> wyznacza komponent realizujący usługę. Stereotyp <<Consumer>> oznacza komponent korzystający z usługi. Do pokazania kontraktów możemy użyć diagramu komponentów UML. Co więcej, kontrakt może służyć do określenia wymagań poza funkcjonalnych. Widok *Integrowanych usług* pokazuje komponenty biorące udział w interakcji między systemami oraz samą wymianę komunikatów między usługodawcami a konsumentami. W celu przedstawienia komponentów stosuje się diagram komponentów UML. Natomiast, samo korzystanie z usługi zazwyczaj wiąże się z procesem integracji. Na przepływ integracyjny składają się mechanizmy mediacji. Fizyczną instalację środowiska wykonawczego systemu oprogramowania można przedstawić w widoku *Wdrożeniowym*. Instalacja wdrożeniowa składa się ze sprzętu i środowiska wykonawczego niezbędnego do uruchomienia opracowanego systemu oprogramowania. Zwykle wykorzystuje się diagramu wdrożeniowy UML. Widok ten przedstawia również rozmieszczenie komponentów oprogramowania na węzłach fizycznych.
Po raz pierwszy model widoków architektonicznych 1+5 został zaprezentowany na Krajowej Konferencji Inżynierii Oprogramowania w roku 2012 organizowanej przez Akademię Górniczo-Hutniczą. Wystąpienie spotkało się z pozytywnym przyjęciem i habilitant został zaproszony do opublikowania zaprezentowanej pracy w czasopiśmie Journal of Theoretical and Applied Computer Science [20]. Model 1+5 został także opisany w książce *„Platformy integracyjne. Zagadnienia wybrane"* wydanej w 2012 roku przez Państwowe Wydawnictwo Naukowe [21]. Ponadto, habilitant pokazał, że model 1+5 jest dostosowany do projektowania rozwiązań w technologii łańcucha bloków. W szczególności widok *Kontraktów* jest predystynowany do projektowania inteligentnych kontraktów technologii łańcucha bloków. Aspekt ten został zaprezentowany po raz pierwszy na konferencji EUROCAST 2019 [22], [23]. Model 1+5 w wersji obejmującej obszary projektowania zarówno integracji systemów informatycznych jak i rozwiązań w technologii łańcucha został przedstawiony w artykule [A7].

IV.4.1. Profil *UML Profile for Messaging Patterns*

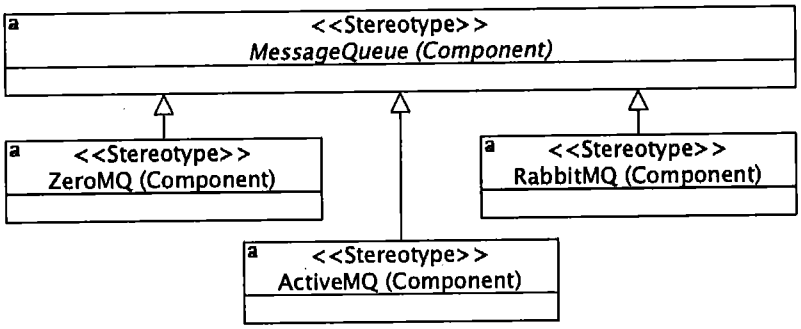
Zaproponowano nowy stereotyp <<IntegratedSystem>> języka UML reprezentujący integrowany system zewnętrzny [A7]. Stereotyp można zastosować do elementu bazowego *UML Actor* na diagramie przypadków użycia UML w widoku *Przypadków użycia*. Ponadto, opracowano dedykowaną ikonę do oznaczania integrowanego systemu zewnętrznego [A3]. Rysunek 2 przedstawia ikonę opracowaną dla stereotypu <<IntegratedSystem>>.



Rysunek 2. Ikona stereotypu integrowanego systemu.

Typy komponentów komunikacyjnych umieszczone się w widoku *Integrowanych usług*. Profil obejmuje dwa typy komponentów komunikacyjnych: kolejki komunikatów i magistrale usług korporacyjnych. W profilu uwzględniono stereotypy dotyczące obu typów komponentów. Stereotypy dotyczące komponentów komunikacyjnych wykorzystują typ elementu *UML Component* jako podstawowy klasyfikator UML. Dla ogólnej kolejki komunikatów zadeklarowano abstrakcyjny stereotyp <<MessageQueue>>. Trzy konkretne stereotypy oznaczają rzeczywiste środowiska kolejek komunikatów: <<ZeroMQ>>, <<RabbitMQ>>, oraz <<ActiveMQ>>.

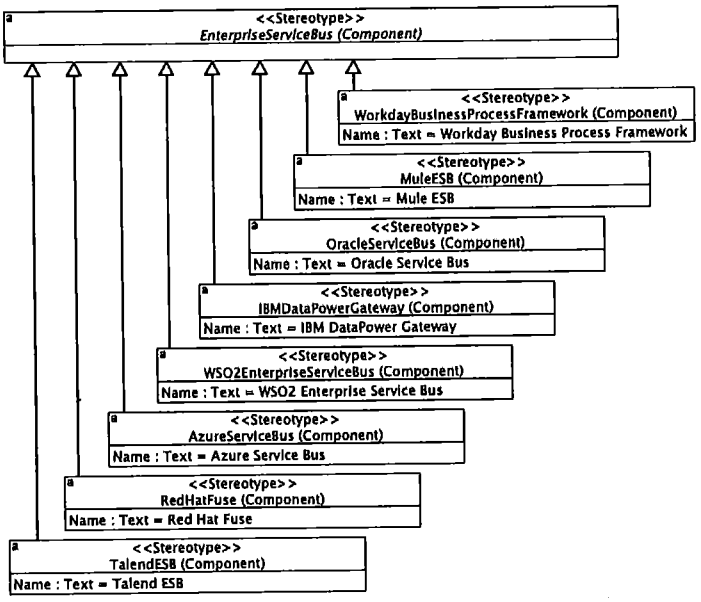
Rysunek 3 przedstawia hierarchię dziedziczenia stereotypów dla kolejek komunikatów.



Rysunek 3. Diagram profilu UML z hierarchią dziedziczenia stereotypów dla kolejek komunikatów.

Poza tym zadeklarowano abstrakcyjny stereotyp <<EnterpriseServiceBus>> dla ogólnej korporacyjnej magistrali usług (ang. Enterprise Service Bus, ESB). Osiem konkretnych stereotypów jest specjalizacją tego abstrakcyjnego i oznacza środowiska faktyczne, m.in.: <<MuleESB>>, <<WSO2EnterpriseServiceBus>>, <<RedHatFuse>>, <<TalendESB>> oraz <<OracleServiceBus>>.

Rysunek 4 pokazuje hierarchię dziedziczenia stereotypów dla korporacyjnych magistral usług.



Rysunek 4. Diagram profilu UML z hierarchią dziedziczenia stereotypów dla korporacyjnych magistral usług.

Tabela 5 przedstawia komplet stereotypów dotyczących konkretnych korporacyjnych magistral usług zawartych w profilu.

Tabela 5. Stereotypy dla korporacyjnych magistral usług.

Stereotyp	Rodzaj	Typ bazowy UML
<<EnterpriseServiceBus>>	Abstrakcyjny	UML Component
<<WSO2EnterpriseServiceBus>>	Konkretny	UML Component
<<WorkdayBusinessProcessFramework>>	Konkretny	UML Component
<<OracleServiceBus>>	Konkretny	UML Component
<<TalendESB>>	Konkretny	UML Component
<<RedHatFuse>>	Konkretny	UML Component
<<MuleESB>>	Konkretny	UML Component
<<AzureServiceBus>>	Konkretny	UML Component
<<IBMDataPowerGateway>>	Konkretny	UML Component

Ponadto istnieją różne protokoły komunikacyjne, szczególnie w obszarze Internetu Rzeczy (ang. Internet-of-Things, IoT). Dlatego też zadeklarowano stereotypy dotyczące najczęściej używanych protokołów przesyłania wiadomości IoT. Nie ma zadeklarowanych ikon dla stereotypów odnoszących się komponentów strukturalnych, ponieważ stereotypy te można stosować w różnym kontekście. Stereotypami dla protokołów oznacza się połączenia pomiędzy

dwoma elementami modelowania. Połączenie, czyli element typu *UML Control Flow* łączy dwie konstrukcje typu *UML Actions*. Połączenia mają standardowy wygląd w języku UML. Tabela 6 przedstawia stereotypy dla protokołów komunikacyjnych.

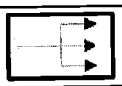
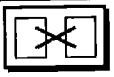
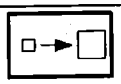
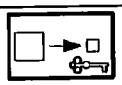
Tabela 6. Stereotypy dla protokołów komunikacyjnych.

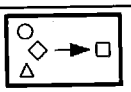
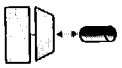
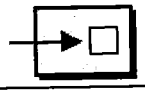
Nazwa	Stereotyp	Typ bazowy UML
Message Queuing Telemetry Transport	<<MQTT>>	UML Control Flow
Advanced Message Queuing Protocol	<<AMQP>>	UML Control Flow
Constrained Application Protocol	<<CoAP>>	UML Control Flow
HyperText Transfer Protocol	<<HTTP>>	UML Control Flow
Extensible Messaging and Presence Protocol	<<XMPP>>	UML Control Flow
Data Distribution Service	<<DDS>>	UML Control Flow

Standardowe mechanizmy przesyłania wiadomości zostały opisane przez Hohpe i Wolfa w [6] i nazwane wzorcami integracji przedsiębiorstw (ang. Enterprise Integration Patterns, EIPs). Pierwsza wersja profilu do modelowania przepływów integracyjnych zawierała zestaw tylko tych mechanizmów mediacji. Profil ten został nazwany *UML for Integration Flows* i opublikowały w artykule [A7]. Projekt tego profilu habilitant wykonał w narzędziu IBM Rational Software Architect i w postaci pliku z rozszerzeniem *.epx został udostępniony w publicznym repozytorium GitHub [15].

Tabela 7 przedstawia stereotypy dla wybranych wzorców EIP.

Tabela 7. Wybrane stereotypy dla wzorców integracji przedsiębiorstw.

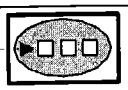
Stereotyp	Typ bazowy UML	Ikona
<<MessageChannel>>	UML Action	
<<Message>>	UML Action	
<<PublishSubscribeChannel>>	UML Action	
<<RequestReply>>	UML Action	
<<MessageTranslator>>	UML Action	
<<ContentEnricher>>	UML Action	
<<ClaimCheck>>	UML Action	

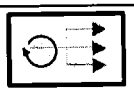
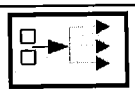
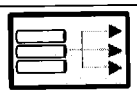
<<MessageRouter>>	UML Action	
<<Normalizer>>	UML Action	
<<GuaranteedDelivery>>	UML Action	
<<DeadLetterChannel>>	UML Action	
<<ChannelAdapter>>	UML Action	
<<MessageEndpoint>>	UML Action	

Uaktualniona wersja tego profilu, uwzględniająca aktualnie opracowane wzorce przesyłania komunikatów została przygotowana w postaci profilu *UML Profile for Messaging Patterns*. Profil został opublikowany w artykule [A4] a jego rozszerzona i zaktualizowana wersja została przedstawiona w artykule [A3]. Habilitant zaprojektował profil w narzędziu Visual Paradigm w postaci pliku z rozszerzeniem *.vpp. Profil został udostępniony w publicznym repozytorium GitHub [8]. Według strony internetowej dla wzorcami integracji przedsiębiorstw istnieje ponad 60 wzorców komunikatów [7]. Dla każdego z nich zadeklarowano dedykowany stereotyp.

W profilu uwzględniono cztery wzorce wynikające z aktualnych prac badawczych. Wszystkie są rozszerzeniami wzorców integracji przedsiębiorstwa: Saga i Publish-Subscribe Channel. Najwięcej specjalizacji zaproponowano dla wzorca Publish-Subscribe Channel. Wprowadzono trzy nowe wersje tego wzorca [3], [4], [5]. Dodatkowo zmodyfikowany wzorzec Saga podnosi integralność transakcji na wyższy poziom [2]. Dla tych nowych wzorców integracji zaproponowano stereotypy: <<ResilientPS>>, <<PairwisePS>>, << SeparatePS >> oraz <<SyncSaga>>. W nazwach nowych stereotypów rozszerzających wzorzec Publish-Subscribe Channel zawarty jest przedrostek charakteryzujący cechę wyróżniającą. Przyrostek jest taki sam. Ponadto, zaprojektowano nowe i unikalne ikony, w kolorystyce wzorców integracji przedsiębiorstwa. Habilitant opublikował pełen opis tych wzorców w publikacji [A4]. Tabela 8 przedstawia stereotypy dla wzorców projektowych będących specjalizacją wzorców Saga oraz Publish-Subscribe Channel.

Tabela 8. Stereotypy dla wzorców projektowych wynikających z aktualnych prac badawczych.

Nazwa wzorca	Stereotyp	Typ bazowy UML	Ikona
Saga for Synchronized Commits	<<SyncSaga>>	UML Action	

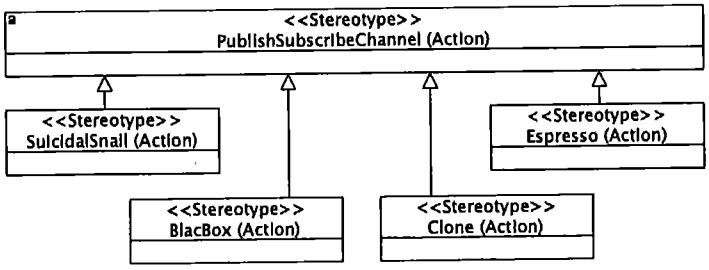
Resilient Publish-Subscribe	<<ResilientPS>>	UML Action	
Pairwise Publish-Subscribe	<<PairwisePS>>	UML Action	
Separate Publish-Subscribe	<<SeparatePS>>	UML Action	

Środowiska dla rozwiązań integracyjnych stale ewoluują i pojawiają się nowe lub ulepszone mechanizmy rozwiązywania istniejących problemów z wymianą komunikatów. Kolejka komunikatów ZeroMQ wnosi do tego obszaru sporo świeżości. Środowisko to definiuje wyspecjalizowane wersje wzorców Publish-Subscribe Channel [24] i Request-Reply [25]. Oryginalne ikony EIP dla wzorców Publish-Subscribe Channel i Request-Reply zostały zmodyfikowane. Ze względu na kolorystykę środowiska ZeroMQ przyjęto czarno-czerwono-biały schemat kolorów dla tych ikon. Habilitant opublikował pełen opis wzorców przesyłania komunikatów dla środowiska ZeroMQ w artykule [A4].

Wprowadzono następujące stereotypy dla wzorców ZeroMQ, które specjalizują wzorzec Publish-Subscribe Channel:

- <<SuicidalSnail>> — wykrywa wolnych abonentów w celu utrzymania określonego maksymalnego opóźnienia.
- <<Espresso>> — monitoruje sieć i natychmiast drukuje wszystkie przychodzące wiadomości.
- <<BlackBox>> — wykrywa abonentów o dużej prędkości i wykorzystuje wielowątkowość do wysyłania i odczytywania wiadomości w oddzielnych wątkach.
- <<Clone>> — utrzymuje wspólny stan wśród grupy klientów korzystających z magazynu klucz-wartość.

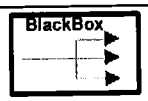
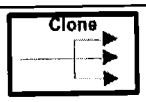
Rysunek 5 przedstawia hierarchię dziedziczenia stereotypów ZeroMQ dla specjalizacji wzorca Publish-Subscribe Channel.



Rysunek 5. Diagram profilu UML z hierarchią dziedziczenia stereotypów ZeroMQ dla specjalizacji wzorca Publish-Subscribe Channel.

Tabela 9 przedstawia stereotypy ZeroMQ wraz z ikonami dla specjalizacji wzorca Publish-Subscribe Channel.

Tabela 9. Stereotypy ZeroMQ dla specjalizacji wzorca Publish-Subscribe Channel

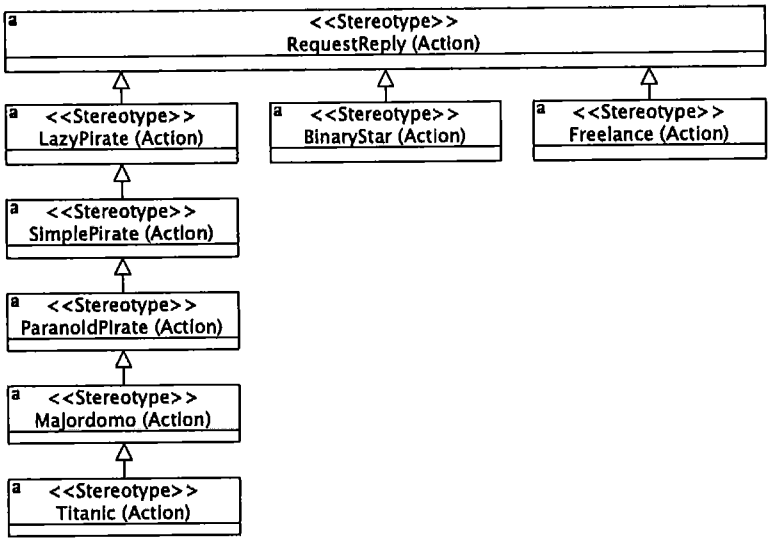
Nazwa wzorca	Stereotyp	Typ bazowy UML	Ikona
Suicidal Snail	<<SuicidalSnail>>	UML Action	
Espresso	<<Espresso>>	UML Action	
Black Box	<<BlackBox>>	UML Action	
Clone	<<Clone>>	UML Action	

Poza tym wprowadzono następujące stereotypy dla wzorców ZeroMQ, które specjalizują wzorzec Request-Reply:

- <<LazyPirate>> — odpytuje gniazdo i odbiera z niego tylko wtedy, gdy jest pewien, że nadeszła odpowiedź. Jeżeli w określonym czasie nie nadeszła żadna odpowiedź, wzorzec ponownie wysyła żądanie. Transakcja zostaje porzucona w przypadku braku odpowiedzi po kilku żądaniach.
- <<SimplePirate>> — rozszerza wzorzec Lazy Pirate o kolejkowe proxy, które pozwala na przejrzystą komunikację z wieloma serwerami.
- <<ParanoidPirate>> — umożliwia niezawodne i niezawodne kolejkowanie, które umożliwia wysyłanie i odbieranie wiadomości w dowolnym momencie, ale wymaga własnego zarządzania kopertami.
- <<Majordomo>> — dodaje nazwę usługi do żądań wysyłanych przez klienta i prosi serwery o zarejestrowanie się w celu uzyskania określonych usług. Dodanie nazw usług powoduje, że wzorzec Paranoid Pirate z kolejki staje się brokerem zorientowanym na usługi.
- <<Titanic>> — przechowuje wiadomości w brokerze wiadomości, aby mieć pewność, że nigdy się nie zgubią.
- <<BinaryStar>> — umieszcza dwa serwery w parze podstawowy-dodatkowy o wysokiej dostępności. W dowolnym momencie aktywny serwer akceptuje połączenia z aplikacji klienckich. Serwer pasywny jest bezczynny, ale serwery monitorują się nawzajem. Jeśli serwer aktywny przestanie działać, serwer pasywny przejmuje funkcję aktywnego.

– <<Freelance>> — tworzy pulę serwerów nazw, więc jeśli jeden przestanie działać, klienci będą mogli połączyć się z innym. W tej architekturze duża grupa klientów łączy się bezpośrednio z kilkoma serwerami w puli. Klienci łączą się z pulą, co jest przeciwieństwem podejścia opartego na brokerze, takiego jak Majordomo, w którym klienci łączą się z brokerem.

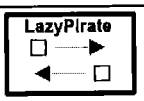
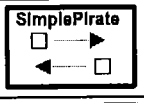
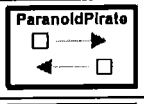
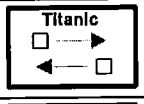
Rysunek 6 przedstawia hierarchię dziedziczenia stereotypów ZeroMQ dla specjalizacji wzorca Request-Reply.



Rysunek 6. Diagram profilu UML z hierarchią dziedziczenia stereotypów ZeroMQ dla specjalizacji wzorca Request-Reply.

Tabela 10 przedstawia stereotypy ZeroMQ wraz z ikonami dla specjalizacji wzorca Request-Reply.

Tabela 10. Stereotypy ZeroMQ dla specjalizacji wzorca Request-Reply.

Nazwa wzorca	Stereotyp	Typ bazowy UML	Ikona
Lazy Pirate	<<LazyPirate>>	UML Action	
Simple Pirate	<<SimplePirate>>	UML Action	
Paranoid Pirate	<<ParanoidPirate>>	UML Action	
Majordomo	<<Majordomo>>	UML Action	
Titanic	<<Titanic>>	UML Action	
Binary Star	<<BinaryStar>>	UML Action	

Freelance	<<Freelance>>	UML Action	

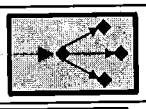
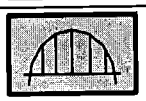
Stereotyp <<Titanic>> oznacza najbardziej wyspecjalizowaną wersję wzorca Request-Reply. Natomiast, najbardziej elastyczny wariant wzorca Request-Reply oznaczono stereotypem <<Freelance>>.

W środowisku Apache Camel zaimplementowane są głównie wzorce integracji przedsiębiorstw EIPs. W tym środowisku zaimplementowany jest praktycznie pełen zakres tych wzorców [26]. Kolejka komunikatów Apache Camel oferuje także dodatkowe własne wzorce:

- <<LoadBalancer>> — umożliwia delegowanie wiadomości do jednego z punktów końcowych przy użyciu różnych polityk równoważenia obciążenia.
- <<ResumableConsumer>> — strategia wznowienia umożliwia pominięcie odczytu i przetwarzania danych, które zostały już zużyte.
- <<ChangeDataCapture>> — umożliwia śledzenie zmian w bazach danych, a następnie pozwala aplikacjom nasłuchiwać zmian w zdarzeniach i odpowiednio reagować.

Tabela 11 przedstawia stereotypy dla wzorców integracji Apache Camel wraz z ikonami. Przyjęto pomarańczowe tło dla ikon związanych ze wzorami dla tego środowiska.

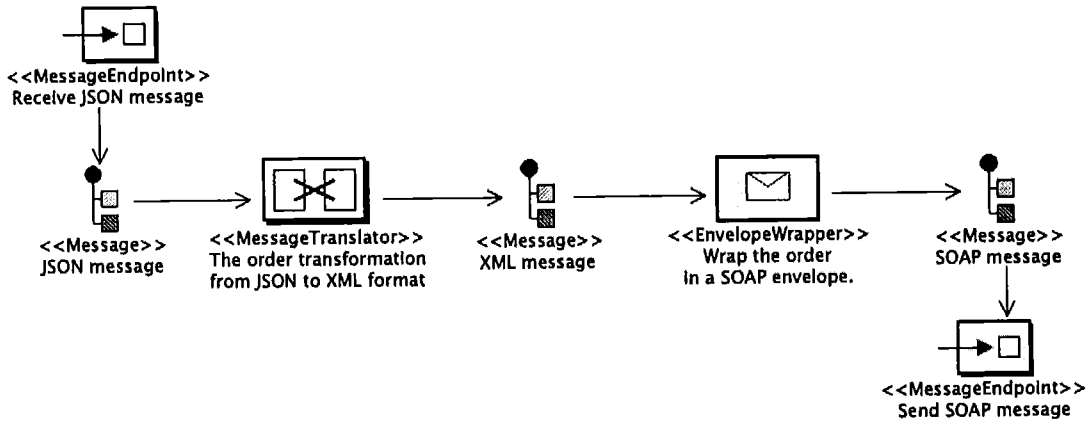
Tabela 11. Stereotypy dla wzorców integracji środowiska Apache Camel.

Nazwa wzorca	Stereotyp	Typ bazowy UML	Ikona
Load Balancer	<<LoadBalancer>>	UML Action	
Resumable Consumer	<<ResumableConsumer>>	UML Action	
Change Data Capture	<<ChangeDataCapture>>	UML Action	

IV.4.2. Diagram przepływów integracyjnych

Współpraca między systemami często realizowana jest w postaci wywoływania usług. Generalnie systemy informatyczne tworzone są niezależnie. Dlatego mogą różnić się w kontekście zastosowanych technologii, protokołów komunikacji, formatów oraz zakresu danych. W celu usprawnienia modelowania komunikacji między systemami zaproponowano *diagram przepływów integracji*, który jest specjalizowaną postacią diagramu aktywności języka UML. Na diagramie tym modelowany jest ciąg działań realizowany w celu przesłania komunikatu z systemu źródłowego do systemu docelowego. Na diagramie tym wykorzystywane są stereotypy z profilu *UML Profile for Messaging Patterns* (*IV.4.1*).

Rysunek 7 przedstawia przepływ integracyjny wykonujący konwersję przesyłanego komunikatu z formatu JavaScript Object Notation (JSON) do XML (Extensible Markup Language), oraz opakowujący go w kopertę protokołu Simple Object Access Protocol (SOAP).



Rysunek 7. Przepływ integracyjny z konwersją komunikatu z formatu JSON na SOAP.

IV.4.3. Metoda modelowania przepływów integracyjnych

W metodzie modelowania przepływów integracyjnych wyróżniono następujące etapy [A3]:

- modelowanie procesów biznesowych - odbywa się to za pomocą diagramów aktywności UML. Często modelowanie biznesowe realizowane jest przy użyciu diagramu procesów BPMN. W metodzie zastosowano UML, aby zachować tę samą notację dla procesów biznesowych i modelowania systemu.
- identyfikacja zadań biznesowych związanych z integracją,
- identyfikacja przypadków użycia biorących udział w integracji,
- modelowanie wszystkich potrzebnych przepływów usług i identyfikacja usług,
- kompozycja przepływu biznesowego z przepływów usług w celu realizacji zadania biznesowego,
- modelowanie komponentów oprogramowania z usługami.

W metodzie zidentyfikowano dwa typy przepływów integracyjnych: usługowy oraz biznesowy [A3]. Habilitant wprowadził następujące definicje przepływu usługowego oraz przepływu biznesowego.

Definicja 1. Przepływ usługowy

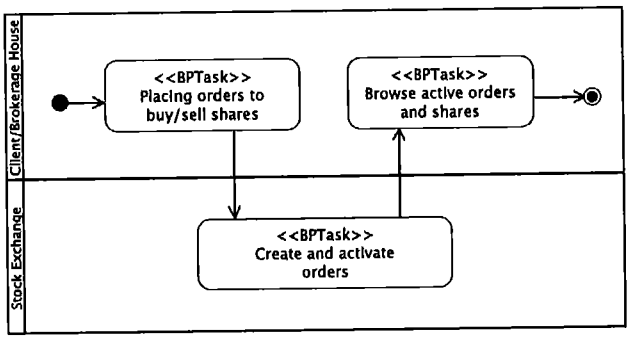
Przepływ usługowy to sekwencja etapów przetwarzania komunikatu, która ma swój początek w systemie źródłowym, przebiega w komponencie integracji i dociera do systemu docelowego w celu wywołania określonej usługi.

Definicja 2. Przepływ biznesowy

Przepływ biznesowy to sekwencja przepływów usługowych uruchamianych na platformie integracyjnej w celu osiągnięcia celu zadania biznesowego.

W artykule [A3] zilustrowano metodę na przykładzie składania zleceń kupna i sprzedaży akcji w domu maklerskim (ang. Brokerage House, BH). Zlecenia złożone w domu maklerskim przekazywane są na Giełdę Papierów Wartościowych (ang. Stock Exchange, SE). W tym miejscu pojawia się komunikacja pomiędzy systemami tych dwóch podmiotów. System SE zarządza zleceniami kupna/sprzedaży. Aby transakcja kupna/sprzedaży mogła zostać zawarta w systemie SE, muszą istnieć dwa odwrotne zlecenia sprzedaży i kupna akcji tego samego podmiotu. Dodatkowo kupujący musi wyrazić zgodę na zapłatę ceny żądanej przez sprzedającego.

Rysunek 8 przedstawia diagram aktywności UML z procesem biznesowym składania zleceń sprzedaży/kupna akcji.

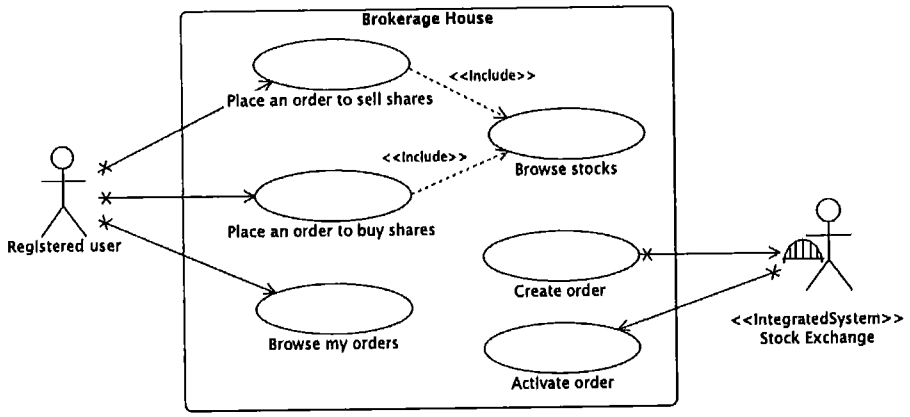


Rysunek 8. Proces biznesowy składania zleceń sprzedaży/kupna.

Realizacja zadania biznesowego *Create and activate orders* wymaga dwukierunkowej komunikacji, a tym samym przepływu biznesowego, który obejmuje co najmniej dwa przepływy usługowe.

Działania w procesie biznesowym mogą mieć różny charakter. Niektóre można zautomatyzować w formie usług lub przepływów integracyjnych. Inne mogą obejmować interakcje z użytkownikiem systemu. Widok *Przypadków użycia* definiuje funkcje systemu ważne dla jego użytkowników, czyli przypadki użycia. Projekty tych funkcji, czyli realizacje przypadków użycia, prezentowane są w widoku *Logicznym*. W kontekście przykładu wysyłania zleceń sprzedaży/kupna istotne jest podkreślenie początku procesu biznesowego. Aby móc przesyłać zlecenia z domu maklerskiego na giełdę, ktoś je musi złożyć. W tym celu najczęściej wykorzystuje się aplikację domu maklerskiego. Musi ona oferować określone funkcje, w tym składanie zleceń kupna lub sprzedaży. Na diagramie przypadków użycia powinniśmy zidentyfikować wszystkie niezbędne przypadki użycia.

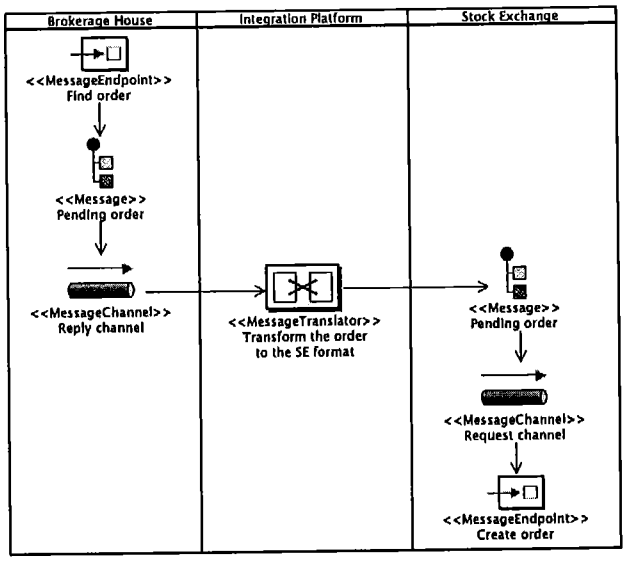
Rysunek 9 przedstawia fragment diagramu przypadków użycia UML dla aplikacji Domu Maklerskiego. Diagram przedstawia między innymi przypadki użycia *Create order* i *Activate order*, które komunikują się z systemem zewnętrznym. Oznaczenie aktora *Stock Exchange* stereotypem <<IntegratedSystem>> określa potrzebę integracji.



Rysunek 9. Diagram przypadków użycia UML dla aplikacji Domu Maklerskiego.

W widoku *Integrowanych usług* należy zamodelować przepływy usługowe dla tych dwóch przypadków użycia: *Create order* i *Activate order*. Dynamiczny aspekt komunikacji realizowany jest poprzez przepływy integracyjne. Na diagramach przepływu integracji komponenty są reprezentowane jako partycje, a usługi są używane jako punkty końcowe. Umieszczając akcje w partycjach, przydzielamy odpowiedzialność komponentom.

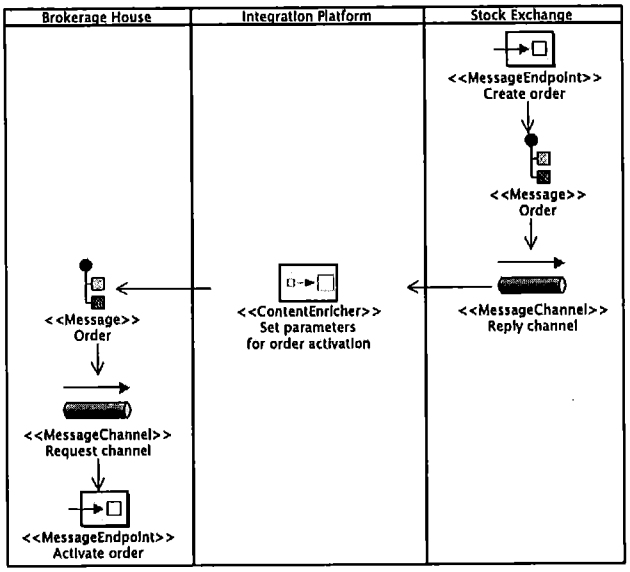
Rysunek 10 przedstawia przepływ integracyjny dla przypadku użycia *Create order*. Przepływ odpowiada za przesłanie zlecenia z aplikacji BH do systemu SE.



Rysunek 10. Przepływ usługowy dla przypadku użycia Create order.

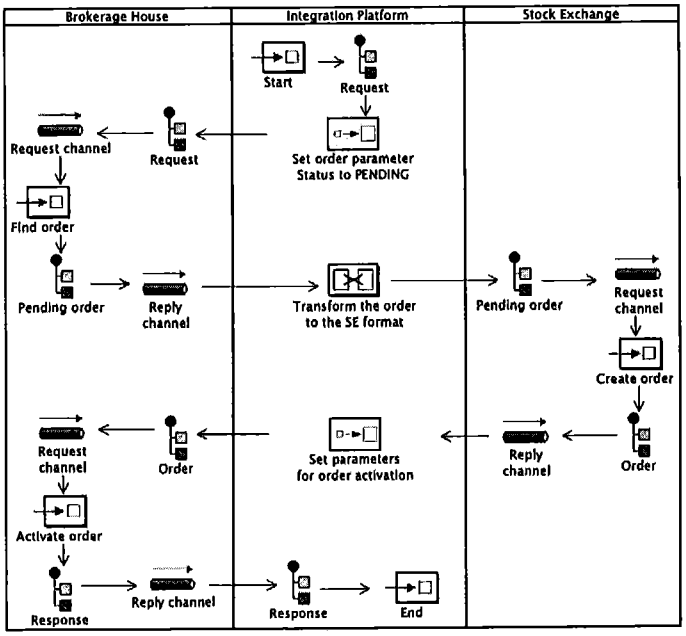
Przepływy kontroli (ang. Control flows) na Rysunek 8 dwukrotnie przekraczają granicę organizacji domu maklerskiego. Zatem w ramach komunikacji powinniśmy zidentyfikować co

najmniej dwa przepływy usług. Pierwszy przypływ płynie z BH do SE - w ramach przypadku użycia *Create order*. Drugi przepływ płynie w odwrotną stronę, wysyła wiadomości z SE do BH - w ramach przypadku użycia *Activate order*. Rysunek 11 przedstawia drugi przepływ integracji, który aktywuje zlecenie w aplikacji BH.



Rysunek 11. Przepływ usługowy dla przypadku użycia Activate order.

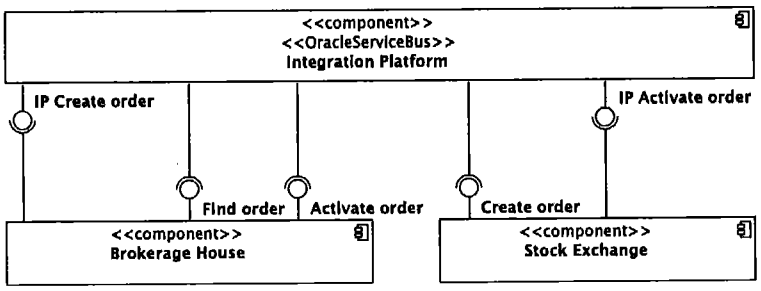
Aby uzyskać pełen obraz przesyłania komunikatów należy jednak zamodelować przepływ biznesowy. W przykładzie składania zleceń kupna/sprzedaży akcji istnieją cztery przepływy usług połączone w logiczny łańcuch kolejnych wezwań do usług. Rysunek 12 przedstawia diagram przepływu integracji dla przepływu biznesowego w kontekście zadania biznesowego *Create and activate orders*.



Rysunek 12. Przypływ biznesowy dla zadania biznesowego "Create and activate orders".

Aspekt strukturalny jest modelowany przy użyciu diagramu komponentów UML i obejmuje komponenty systemu oraz komponent kolejki komunikatów lub szyny usług. Diagram komponentów UML przedstawia dwie ważne informacje: komponenty, które ze sobą współdziałają, oraz usługi świadczone przez każdy z nich. Usługi modelowane są za pomocą interfejsów realizowanych przez komponenty.

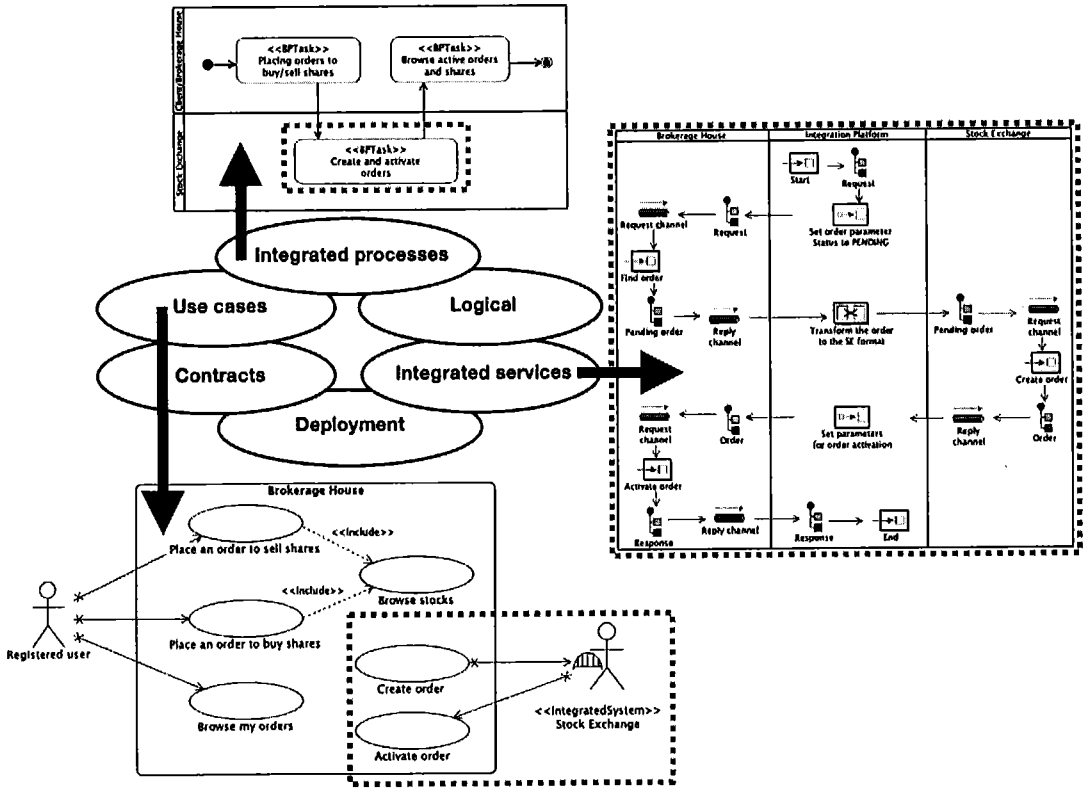
Rysunek 13 przedstawia diagram komponentów UML dla integracji aplikacji domu maklerskiego i giełdy papierów wartościowych.



Rysunek 13. Diagram komponentów UML dla integracji aplikacji domu maklerskiego i giełdy papierów wartościowych.

Dopiero modelując pełny przepływ biznesowy, możemy zidentyfikować wszystkie niezbędne usługi w komponentach zaangażowanych w komunikację.

Rysunek 14 przedstawia zależności między widokami *Integrowanych procesów*, *Przypadków użycia* oraz *Integrowanych usług*, w kontekście modelowania przepływów integracyjnych.



Rysunek 14. Zależności między widokami architektonicznymi w opisie przepływów integracyjnych.

Habilitant przedstawił pełen opis metody modelowania przypływów integracyjnych, wraz z definicjami typów przepływów, w artykule [A3].

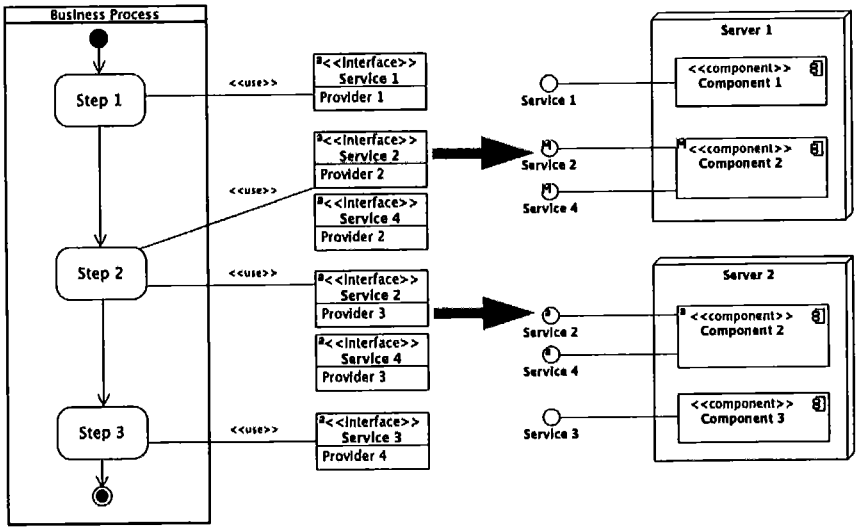
IV.4.4. Przegląd metod optymalizacji wykonania procesów biznesowych

Artykuł [A8] stanowi przegląd metod optymalizacji uruchamiania procesów biznesowych. Natomiast, w kontekście wkładu do habilitacji pokazuje istotność opisu architektonicznego rozwiązań w architekturze usługowej w następujących widokach architektonicznych: *Integrowanych procesów*, *Integrowanych usług*, *Logicznym* oraz *Wdrożeniowym*.

Poszczególne usługi mogą być dostarczane przez różnych dostawców, czyli komponenty wdrożone na odrębnych fizycznych maszynach.

Należy podkreślić, że widok *Integrowanych procesów* został włączony do opisu architektonicznego przez Habilitanta. Nazwa tego widoku także jest autorską propozycją Habilitanta. Usługi identyfikowane są w autorskim widoku *Integrowanych usług*. Wszystkie projektowane elementy oprogramowania z widoku *Logicznego* muszą znaleźć swoje umiejscowienie w widoku *Wdrożeniowym* w postaci komponentów.

Rysunek 15 przedstawia usługi wykorzystywane przez kroki procesu biznesowego.



Rysunek 15. Usługi realizujące kroki procesu biznesowego z ich przypisaniem do dostawców.

Przedstawione w tym artykule spojrzenie potwierdza potrzebę opisu od poziomu procesu biznesowego do poziomu wdrożeniowego. Zapewnia to kompletność opisu z punktu widzenia opisywanych poziomów abstrakcji dla projektu systemu informatycznego.

Zebranie zestawu istotnych artykułów oraz wstępna analiza danych została wykonana przez współautora artykułu [A8].

IV.4.5. Profil *UML Profile for Distributed Ledger Deployment*

W konstrukcji profilu Habilitant wykorzystał mechanizmy rozszerzalności języka UML jakimi są stereotypy (ang. stereotype) oraz wartości oznakowane (ang. tagged values). Wszystkie zaproponowane stereotypy i oznaczone wartości opisują nowe konstrukcje semantyczne UML dla opisu widoku *Wdrożeniowego* sieci łańcucha bloków. Habilitant użył stereotypów do przedstawienia węzłów, usług i protokołów komunikacyjnych charakterystycznych dla sieci łańcucha bloków R3 Corda wersji 4.3.

Tabela 12 przedstawia stereotypy uwzględnione w profilu *UML Profile for Distributed Ledger Deployment.*

Tabela 12. Stereotypy dla węzłów, usług oraz protokołów komunikacyjnych sieci łańcucha bloków Corda.

Stereotyp	**Rodzaj elementu modelowego**	**Typ bazowy UML**
<<CordaNode>>	Węzeł	Node
<<DLTNode>>	Węzeł	Node
<<OracleNode>>	Węzeł	Node
<<NotaryNode>>	Węzeł	Node
<<NetworkMapNode>>	Węzeł	Node
<<permissioningService>>	Usługa	Artifact
<<networkMapService>>	Usługa	Artifact
<<notaryService>>	Usługa	Artifact
<<oracleService>>	Usługa	Artifact
<<identityService>>	Usługa	Artifact
<<supportService>>	Usługa	Artifact
<<HTTPS>>	Protokół komunikacyjny	Generic Connection
<<AMQP/TLS>>	Protokół komunikacyjny	Generic Connection

Dla konfiguracji pojedynczego węzła można wykorzystać parametry wdrożeniowe. Każdy parametr ma swoją nazwę i wartość. Zastosowano wartości oznaczone do modelowania parametrów wdrożeniowych węzłów sieci łańcucha bloków.

Tabela 13 przedstawia wartości oznakowane przypisane parametrom wdrożeniowym węzłów oznaczonych stereotypem <<CordaNode>>.

Tabela 13. Wartości oznakowane dla stereotypu <<CordaNode>>.

Wartość oznakowana	Typ danych	Wartość domyślna
p2pAddress	Tekst	Nie zdefiniowana
additionalP2PAddresses	Tekst	Nie zdefiniowana
attachmentCacheBound	Liczba całkowita	1024
messagingServerAddress	Tekst	Nie zdefiniowana
myLegalName	Tekst	Nie zdefiniowana
detectPublicIp	Logiczny	Fałsz
flowMonitorPeriodMillis	Liczba całkowita	60
flowTimeout.timeout	Liczba całkowita	30
flowTimeout.maxRestartCount	Liczba całkowita	6

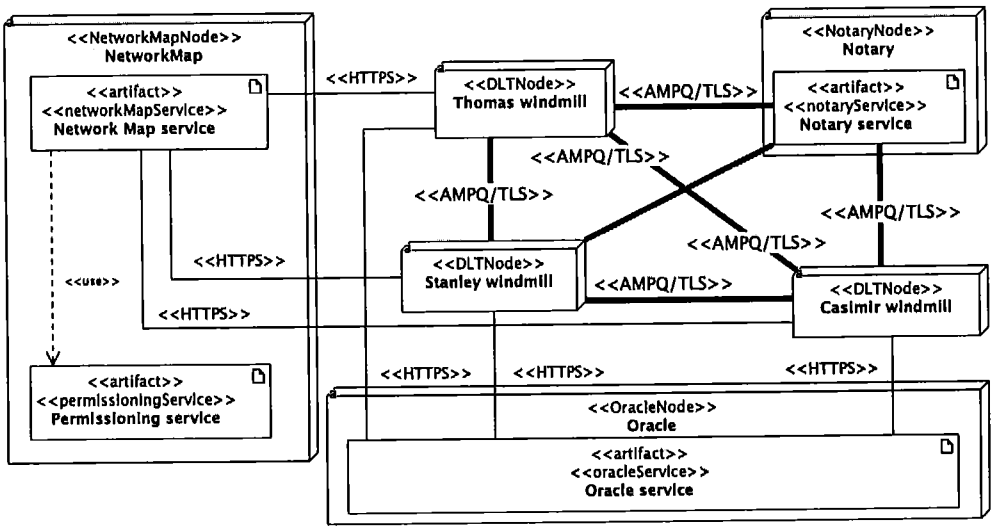
Pełen zestaw obejmuje 52 parametry wdrożeniowe węzła sieci łańcucha bloków R3 Corda. Wszystkie aktualne parametry wdrożeniowe zostały uwzględnione w postaci wartości oznakowanych. W stereotypie <<NotaryNode>> zdefiniowano dodatkowy zestaw parametrów wdrożeniowych charakterystycznych tylko dla tego typu węzła.

Tabela 14 przedstawia wartości oznakowane dla parametrów wdrożeniowych stereotypu <<NotaryNode>>.

Tabela 14. Wartości oznakowane dla stereotypu <<NotaryNode>>.

Wartość oznakowana	Typ danych	Wartość domyślna
notary.validating	Logiczny	Fałsz
notary.serviceLegalName	Tekst	Nie zdefiniowana
notary.dftSMaRt.clusterAddresses	Tekst	Nie zdefiniowana
notary.raft clusterAddresses	Tekst	Nie zdefiniowana
notary.raft.nodeAddress	Tekst	Nie zdefiniowana
notary.bftSMaRt.replicaId	Tekst	Nie zdefiniowana

Rysunek 16 przedstawia diagram wdrożeniowy UML dla środowiska testowego fragmentu sieci łańcucha bloków dla systemu wymiany energii odnawialnej.



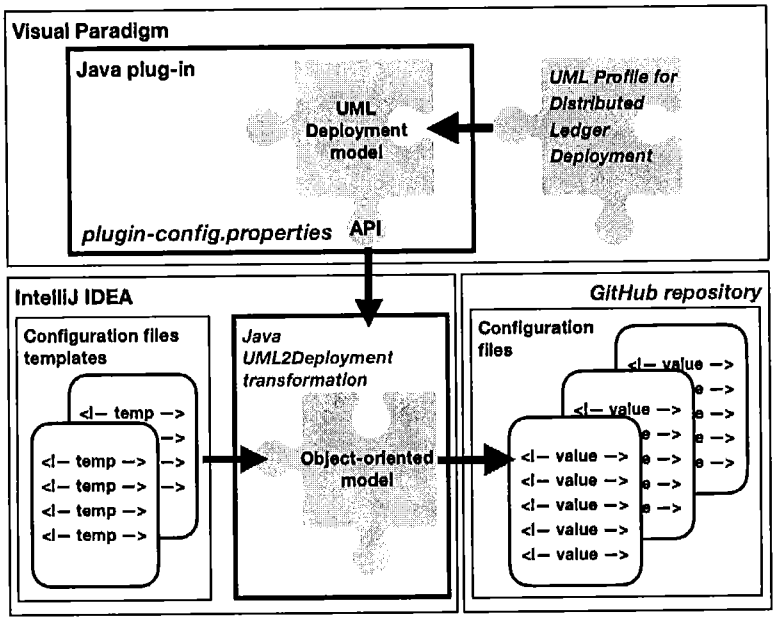
Rysunek 16. Diagram wdrożeniowy UML sieci łańcucha bloków dla systemu wymiany energii odnawialnej.

Profil *UML Profile for Distributed Ledger Deployment* został przedstawiony przez Habilitanta w artykule [A9]. Profil został zaprojektowany w na tyle elastyczny sposób, że możliwe było jego rozszerzenie, tak aby obejmował zestaw parametrów konfiguracyjnych węzłów sieci łańcucha bloków Corda od wersji 4.3 do wersji 4.6. Zaktualizowany profil został wykorzystany w transformacji do generacji konfiguracji wdrożeniowej sieci łańcucha bloków opisanej w punkcie IV.4.6.

IV.4.6. Transformacja do generacji konfiguracji wdrożeniowej sieci łańcucha bloków

Zastosowano także inżynierię sterowaną modelami (ang. Model-Driven Engineering) do automatyzacji generowania konfiguracji wdrożeniowych sieci łańcucha bloków z modeli wdrożeniowych UML. Zastosowanie tego typu transformacji zapewnia spójność opisu architektonicznego systemu informatycznego z faktycznie uruchomioną jego konfiguracją. Jest to o tyle istotne, że każdy węzeł sieci łańcucha bloków może mieć inną konfigurację wdrożeniową. Transformacja ta umiejscowiona jest w widoku *Wdrożeniowym* modelu 1+5. Habilitant przy projektowaniu transformacji zastosował kilka zasad architektonicznych. Przede wszystkim cecha modularności spowodowała, że rozwiązanie zostało podzielone na moduły: *UML Profile for Distributed Ledger Deployment*, *UML Deployment model*, *Object-oriented model*, *Configuration files templates*, *Configuration files*.

Rysunek 17 przedstawia komponenty wchodzące w skład przygotowanego rozwiązania MDE.



Rysunek 17. Transformacja UML2Deployment do generacji konfiguracji wdrożeniowych węzłów sieci łańcucha bloków

Uwzględnienie cechy rozdzielenia odpowiedzialności skutkowało utworzeniem dwóch osobnych projektów Java: aplikacji transformacyjnej, która przekształca model wdrożenia UML w skrypty konfiguracyjne wdrożenia oraz wtyczki, która integruje transformację z narzędziem do modelowania Visual Paradigm.

Aby uzyskać możliwość zmiany wersji środowiska R3 Corda bez konieczności zmiany transformacji wykorzystano dwa następujące elementy: szablony plików konfiguracyjnych oraz *UML Profile for Distributed Ledger Deployment*. Obydwa elementy posługują się tym samym zestawem stereotypów i oznakowanych wartości. Zestaw ten jest zgodny ze specyfikacją wersji środowiska R3 Corda w wersjach od 4.3 do 4.6. Transformacja wypełnia tylko pasujące elementy w plikach konfiguracyjnych. Transformacja została zaprojektowana w ten sposób, że wystarczy dostarczyć dla nowej wersji profilu odpowiednie szablony plików konfiguracyjnych, aby wygenerować pliki konfiguracyjne dla nowej wersji środowiska R3 Corda. Daje to możliwość zarządzania konfiguracją pomiędzy kolejnymi wersjami platformy także w przypadku zmiany struktury plików konfiguracyjnych.

Z technicznego punktu widzenia mechanizm szablonów zapewnia niezależność od dostawcy środowiska łańcucha bloków. Obecnie nie ma standardów w tym obszarze, a dostawcy platform zarządzają konfiguracją wdrożeniową w różny sposób. W przypadku dostawcy zarządzającego konfiguracją wdrożeniową za pomocą plików konfiguracyjnych opracowane rozwiązanie będzie działać poprawnie przy uwzględnieniu odpowiedniego szablonu oraz dostarczeniu dedykowanego profilu UML. Natomiast, zaproponowany mechanizm szablonów może nie być

odpowiedni przy odmiennym sposobie zarządzania konfiguracją wdrożeniową w innych platformach łańcucha bloków.

Kolejną zastosowaną regułą architektoniczną jest prostota. W rezultacie zidentyfikowano kilka modułów z prostymi zasadami przejścia między nimi. Źródłem transformacji jest *UML Deployment model* z zastosowanymi stereotypami oraz wartościami oznakowanymi z profilu *UML Profile for Distributed Ledger Deployment.* Użyto interfejsu programowania aplikacji (ang. Application Programming Interface, API) narzędzia do modelowania Visual Paradigm, aby uzyskać pełny zestaw węzłów z określonymi wartościami oznakowanymi. Zbiór ten jest przechowywany w *Object-oriented model.* Następnie aplikacja *Java UML2Deployment transformation* odczytuje odpowiednie szablony plików konfiguracyjnych i generuje wdrożeniowe pliki konfiguracyjne. Transformacja ta generuje plik Gradle Groovy DSL (*deployNodesTask.gradle*). Plik ten zawiera wypełnione zadanie Gradle z wymaganymi parametrami konfiguracyjnymi wdrożenia sieci łańcucha bloków. Dla każdego węzła transformacja generuje również plik z pełnym zestawem parametrów wdrożenia.

Zaprojektowana transformacja działa według następującego algorytmu:

- Krok 1a. Użytkownik wywołuje opcję *Generate Node Configuration*, aby wygenerować konfigurację wszystkich węzłów dla wybranego pakietu UML.
- Krok 1b. Użytkownik wywołuje opcję *Generate Node Configuration*, w celu wygenerowania konfiguracji dla pojedynczego węzła,
- Krok 2. Użytkownik wybiera folder docelowy dla wygenerowanych plików konfiguracyjnych.
- Krok 3a. Transformacja zbiera wszystkie węzły UML, które znajdują się w wybranym pakiecie i buduje listę obiektów implementujących interfejs *INode*.
- Krok 3b. Transformacja wybiera węzeł UML i tworzy listę zawierającą jeden obiekt, który implementuje interfejs *INode*.
- Krok 4. Dla każdego obiektu na liście transformacja tworzy obiekt Java odpowiedniej klasy, w oparciu o zastosowany stereotyp, np. obiekt klasy DLTCordaNode dla <<DLTNode>> stereotyp. Transformacja tworzy kolekcję obiektów Collection<CordaNode>.
- Krok 5. Transformacja odczytuje plik szablonu dla zadania Gradle *deployNodes*.
- Krok 6. Dla każdego obiektu Java z kolekcji transformacja:
  - Krok 6.1. odczytuje szablon dla bloku Gradle *node*,
  - Krok 6.2. wypełnia wartości w szablonie bloku Gradle *node* wartościami przechowywanymi w obiekcie Java,
  - Krok 6.3. dodaje wygenerowany blok Gradle *node* do szablonu zadania Gradle *deployNodes*,
  - Krok 6.4. odczytuje szablon pliku konfiguracyjnego węzła,

- Krok 6.5. wypełnia wartości w szablonie pliku konfiguracyjnego węzła wartościami przechowywanymi w obiekcie Java,
- Krok 6.6. zapisuje plik konfiguracyjny pojedynczego węzła w wybranym miejscu docelowym.

– Krok 7. Transformacja zapisuje zadanie Gradle *deployNodes* we właściwym miejscu docelowym.
– Krok 8. Transformacja informuje użytkownika o zakończonej generacji.

Przykład diagramu wdrożeniowego UML dla środowiska testowego fragmentu sieci łańcucha bloków dla systemu wymiany energii odnawialnej przedstawiono w sekcji IV.4.6 (Rysunek 16). Taki diagram z zastosowanymi stereotypami i wartościami oznakowanymi stanowi element źródłowy transformacji - *UML Deployment model.* Transformacja generuje pliki konfiguracyjne korzystając z *Object-oriented model* oraz plików szablonów.

Rysunek 18 przedstawia kod źródłowy metody wpisującej ustalone wartości oznakowane w *UML Deployment model* do plików konfiguracyjnych

```
private String populateTemplateWithTags(String template,
                        Map<String, Object> tagsAndValues) {
 for (Map.Entry<String, Object> entry
                                : tagsAndValues.entrySet()) {
  String key = entry.getKey();
  Object value = entry.getValue();
  String tag = TemplateTags.getInstance().buildTemplateTag(key);
  template = template.replace(tag, getValueAsString(tag, value));
 }
 return template;
}
```

Rysunek 18. Metoda wprowadzająca ustalone w modelu wdrożeniowym wartości parametrów do pliku konfiguracyjnego.

Rysunek 19 przedstawia fragment wygenerowanej konfiguracji dla węzła sieci łańcucha bloków.

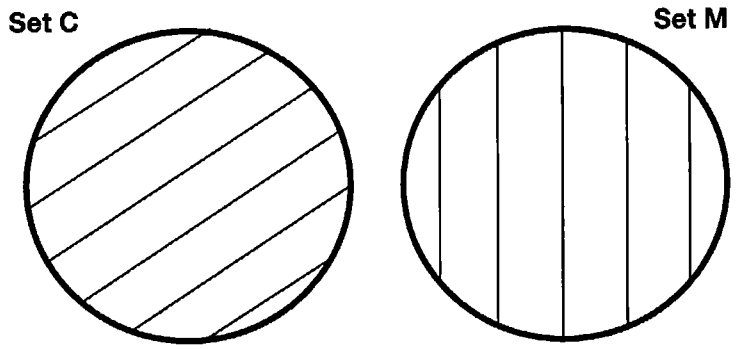
```
myLegalName = "O=Windmill Thomas,L=Gdynia,C=PL"
additionalP2PAddresses =
   <!--additionalP2PAddresses-->
attachmentContentCacheSizeMegaBytes = 10
attachmentCacheBound = 1024
blacklistedAttachmentSigningKeys =
   <!--blacklistedAttachmentSigningKeys-->
cordappSignerKeyFingerprintBlacklist =
   <!--cordappSignerKeyFingerprintBlacklist-->
crlCheckSoftFail = True
custom = {
    jvmArgs = -Xms256m -Xmx768m
}
database = {
    transactionIsolationLevel = REPEATABLE_READ
    exportHibernateJMXStatistics = False
}
```

Rysunek 19. Wygenerowany plik konfiguracji wdrożeniowej pojedynczego węzła sieci łańcucha bloków.

Element docelowy transformacji (*deployNodesTask.gradle*) zawiera konfiguracje wdrożeniowe wszystkich węzłów wchodzących w skład sieci łańcucha bloków rozwiązania. Aplikację transformacji zaimplementował współautor artykułu [A9].

Ponadto, habilitant opracował metodę walidacji transformacji. Mechanizm ten polega na porównaniu element źródłowego z docelowym. Do walidacji przyjęto następujące założenia: elementem źródłowym jest węzeł UML w modelu wdrożenia UML, a elementem docelowym jest plik konfiguracyjny wdrożenia dla tego węzła UML. Obydwa te elementy składają się na zbiór parametrów. Źródło obejmuje zbiór M, który zawiera oznaczone wartości $m \in M$. Element docelowy zawiera zbiór C, który składa się z parametrów konfiguracyjnych wdrożenia $c \in C$. Metoda zakłada niezależność mechanizmu walidacji od kodu źródłowego transformacji. Aplikację walidującą zaimplementował współautor artykułu [A9].

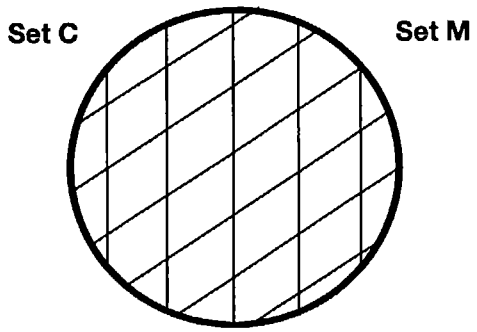
Rysunek 20 przedstawia zbiory parametrów podlegające walidacji.



Rysunek 20. Zbiory parametrów wdrożenia elementów źródłowego oraz docelowego transformacji.

Przecięcie dwóch zbiorów C i M oznaczane jest przez $C \cap M$ i jest zbiorem zawierającym wszystkie elementy C, które również należą do M (lub równoważnie, wszystkie elementy M, które również należą do C). Należało sprawdzić, czy przecięcie tych dwóch zbiorów spełnia następujące równanie: $C \cap M = C = M$.

Aby to zrobić, należy sprawdzić, czy te dwa zbiory zawierają te same elementy o tych samych wartościach (Rysunek 21).



Rysunek 21. Przecięcie zbiorów C i M, gdy parametry w elementach docelowym i źródłowym są zgodne.

Warunki sprawdzenia przecięcia zbiorów $C \cap M$ obejmują przedstawione następujące wzory.

Liczność obu zbiorów powinna być taka sama: $|C| = |M|$.

W celu porównania należy zdefiniować elementy obu zbiorów.

Parametr konfiguracyjny wdrożenia c jest parą uporządkowaną: $c = (n^c, v^c)$, gdzie:

n^c - nazwa parametru konfiguracyjnego wdrożenia c,

v^c - wartość parametru konfiguracyjnego wdrożenia c.

Wartość oznakowana m jest parą uporządkowaną: $m = (n^m, v^m)$, gdzie:

n^m - nazwa oznaczonej wartości m,

v^m - wartość oznaczonej wartości m.

Dla każdego parametru konfiguracyjnego wdrożenia $c \in C$ powinna być odpowiadająca wartość oznakowana $m \in M$, która ma tę samą nazwę i wartość:

$$\bigwedge_{c \in C} \bigvee_{m \in M} (n^c = n^m) \wedge (v^c = v^m)$$

To samo musi być spełnione z przeciwnej strony. Dla każdej wartości oznakowanej $m \in M$ powinien istnieć odpowiadający parametr konfiguracyjny wdrożenia $c \in C$, który ma tę samą nazwę i wartość:

$$\bigwedge_{m \in M} \bigvee_{c \in C} (n^m = n^c) \wedge (v^m = v^c)$$

Należy sprawdzić względne uzupełnienie zbioru C w M, oznaczone jako $M \setminus C$, aby sprawdzić, czy zbiór elementów w M, ale nie w C jest pusty: $M \setminus C = \emptyset$.

Należy także przeprowadzić odwrotne sprawdzenie względnego dopełnienia $C \setminus M$, aby sprawdzić, czy zbiór elementów w C, ale nie w M jest pusty: $C \setminus M = \emptyset$.

Na poziomie projektowym opracowano osobną aplikację walidacyjną, która nie wykorzystuje żadnego elementu aplikacji transformacyjnej. Aplikacja sprawdza dwie kolekcje typu HashMap. Pierwsza zawiera obiekty odpowiadające parametrom konfiguracyjnym wdrożenia. Natomiast druga zawiera obiekty, które odwołują się do wartości oznakowanych węzła w modelu wdrożeniowym UML.

Wszystkie trzy zaprojektowane aplikacje zostały udostępnione w publicznie dostępnych repozytoriach GitHub. Kod źródłowy aplikacji do transformacji modelu wdrożeniowego w pliki konfiguracyjne został umieszczona w repozytorium [12]. Ponadto, aplikacja wtyczki do Visual Paradigm Enterprise dostępna jest w repozytorium [13]. Natomiast, kod źródłowy zaprojektowanej aplikacji do walidacji transformacji został umieszczony w repozytorium [14].

IV.4.7. Wzorzec projektowania inteligentnych kontraktów

W obszarze technologii łańcucha bloków habilitant skupił się na inteligentnych kontraktach. Umiejscowił je w widoku architektonicznych *Kontraktów*. Celem habilitanta było zaproponowanie wzorca projektowania inteligentnych kontraktów umożliwiającego ponowne wykorzystanie reguł weryfikacji. Istotnym było także zapewnienie szybkiej weryfikacji transakcji związanych z inteligentnym kontraktem. Habilitant rozpatrywał także możliwość weryfikacji różnych typów transakcji. Dlatego jednym z celów było uzyskanie możliwości dynamicznej rekonfiguracji inteligentnego kontraktu w zależności od typu weryfikowanej transakcji.

Do konstrukcji wzorca *Smart Contract Design Pattern* (SCDP) wykorzystano mechanizmy programowania obiektowego. Habilitant zaproponował wzorzec umożliwiający rekonfigurację inteligentnego kontraktu z ponownym użyciem reguł weryfikacji. Habilitant przedstawił wzorzec SCDP w artykule [A6]. W artykule tym habilitant wprowadził definicje następujących pojęć: reguła weryfikacji, konfiguracja inteligentnego kontraktu, wyrażenie ewaluacji.

Definicja 3. Reguła weryfikacji

Reguła weryfikacji to pojedynczy warunek nałożony na inteligentny kontrakt. Inteligentne kontrakty mogą obejmować wiele reguł weryfikacji. Aby transakcja mogła zostać zrealizowana, muszą zostać spełnione wszystkie reguły weryfikacji składające się na inteligentny kontrakt.

Definicja 4. Konfiguracja inteligentnego kontraktu

Konfiguracja inteligentnego kontraktu to uporządkowana lista reguł weryfikacji. Reguły weryfikacji nie mogą się powtarzać na liście. Aby transakcja mogła zostać zrealizowana, muszą zostać spełnione wszystkie reguły weryfikacji znajdujące się na liście zgodnie z wyrażeniem ewaluacji.

Definicja 5. Wyrażenie ewaluacji

Wyrażenie ewaluacji to wyrażenie logiczne zawierające reguły weryfikacji i operatory logiczne, które zwraca pojedynczą wartość logiczną.

Wzorzec projektowy SCDP wprowadza jawną deklarację reguł weryfikacji w postaci klas. W celu modelowania inteligentnego kontraktu habilitant zaproponował profil UML, *UML Profile for Smart Contracts*. Profil został udostępniony w repozytorium [10].

W profilu *UML Profile for Smart Contracts* wyodrębniono następujące stereotypy:

- <<AbstractSContract>> - stereotyp używany jest do oznaczania abstrakcyjnej, ogólnej klasy inteligentnych kontraktów. Klasa ta jest nadklasą dla wszystkich typów konkretnych inteligentnych kontraktów.

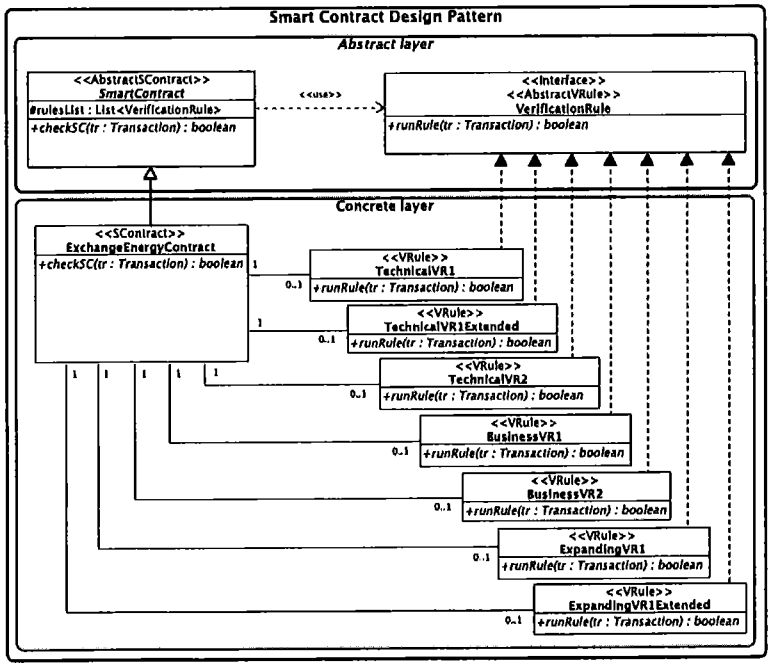
- <<AbstractVRule>> - stereotyp używany jest do oznaczania abstrakcyjnego, ogólnego interfejsu reguły weryfikacji. Interfejs musi być zaimplementowany poprzez konkretne klasy reguł weryfikacji.
- <<SContract>> - stereotyp używany jest do oznaczania konkretnych klas inteligentnych kontraktów.
- <<VRule>> - stereotyp służy do oznaczania klas konkretnych reguł weryfikacji.

Tabela 15 przedstawia stereotypy zaproponowane dla elementów składowych wzorca projektowania inteligentnego kontraktu.

Tabela 15. Stereotypy opisu inteligentnego kontraktu.

Stereotyp	Typ bazowy UML	Warstwa wzorca
<<AbstractSContract>>	Class	Abstrakcyjna
<<AbstractVRule>>	Interface	Abstrakcyjna
<<SContract>>	Class	Konkretna
<<VRule>>	Class	Konkretna

Rysunek 22 przedstawia elementy abstrakcyjne wzorca SCDP oraz przykładem inteligentnego kontraktu w warstwie konkretnej.



Rysunek 22. Warstwy wzorca projektowego Smart Contract Design Pattern.

Obiekty reguł weryfikacji przechowywane są w kolekcji *rulesList*. Konstrukcja wzorca kładzie nacisk na kolejność przechowywania obiektów reguł weryfikacji na liście. Oceniana jest pełna lista reguł, chyba że któraś z nich nie jest spełniona. W takim przypadku weryfikacja zostaje

przerwana i transakcja nie zostaje zrealizowana. Takie podejście może skrócić czas oceny inteligentnego kontraktu. Wzorzec zapewnia ponowne użycie klas reguł weryfikacji. Natomiast w implementacji wzorca warstwa abstrakcyjna nie jest w pełni niezależna od konkretnych inteligentnych kontraktów. W deklaracji klasy abstrakcyjnej *SmartContract* należy podać klasy, które mogą ją rozszerzać. Z jednej strony zabezpiecza to przed dziedziczeniem przez klasy niepowołane, ale z drugiej strony przy implementacji konkretnego inteligentnego kontraktu wymusza modyfikację klasy abstrakcyjnej. Kod źródłowy 1 przedstawia implementację klasy abstrakcyjnej inteligentnego kontraktu *SmartContract*.

Kod źródłowy 1. Klasa abstrakcyjnego inteligentnego kontraktu SmartContract.

```
package pl.gdynia.amw.scdp.contracts;
import pl.gdynia.amw.scdp.Transaction;
import pl.gdynia.amw.scdp.rules.VerificationRule;
import java.util.List;

public abstract sealed class SmartContract permits ExchangeEnergyContract, BuyEnergyFromGrid {
   // list of verification rules
   protected List<VerificationRule> rulesList;
   // verification of the smart contract
   public abstract boolean checkSC(Transaction tr);
}
```

Reguły weryfikacji konstruowane są w postaci klas. Kod źródłowy 2 przedstawia klasę konkretnej reguły weryfikacji *BusinessVR1*.

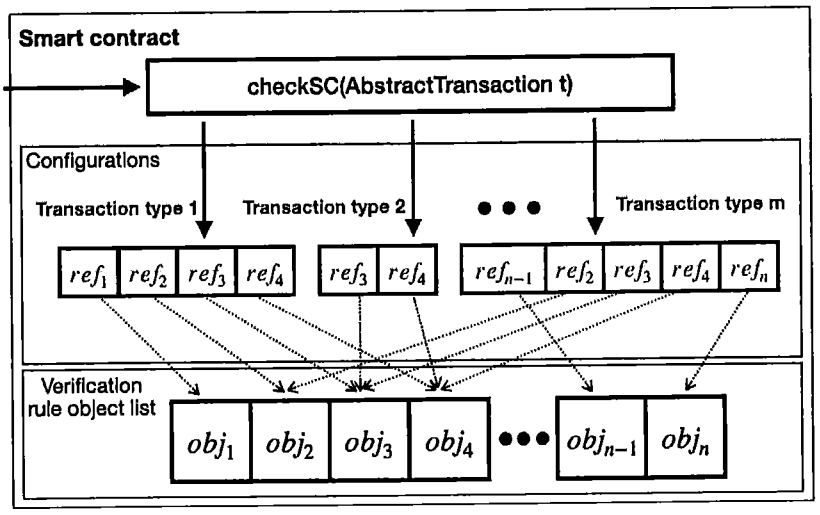
Kod źródłowy 2. Klasa konkretnej reguły weryfikacji BusinessVR1.

```
package pl.gdynia.amw.scdp.rules.exchangeEnergy;
import org.jetbrains.annotations.NotNull;
import pl.gdynia.amw.scdp.Transaction;
import pl.gdynia.amw.scdp.rules.VerificationRule;

public final class BusinessVR1 implements VerificationRule {
   @Override
   public boolean runRule(@NotNull Transaction t){
      if (t.getQuantity() > 0) {
         System.out.println(``BusinessVR1 - quantity > 0``);
         return true; }
      else { System.out.println(``BusinessVR1 - quantity <= 0``);
         return false; }
   }
}
```

Zaletą wzorca SCDP było wyodrębnienie reguł weryfikacji z wewnętrznej implementacji inteligentnych kontraktów oraz zapewnienie ponownego ich użycia pomiędzy konfiguracjami inteligentnego kontraktu, ale także pomiędzy inteligentnymi kontraktami.

Habilitant zaproponował udoskonalony wzorzec projektowania inteligentnych kontraktów. W konstrukcji wzorca projektowania inteligentnych kontraktów AdapT v2.0 wykorzystano zarówno mechanizmy programowania obiektowego jak i funkcyjnego. Habilitant przedstawił wzorzec projektowy AdapT v2.0 w artykule [A2]. Obecnie rozwijane wzorce projektowe inteligentnych kontraktów nie uwzględniają tematyki ich rekonfiguracji, w szczególności możliwości dostosowania do różnego rodzaju przetwarzanych transakcji. Inteligentny kontrakt może wykonywać tę samą operację dla logicznie spójnych, ale różnych typów transakcji. Może to obejmować np. transakcje łańcucha bloków realizowane wewnątrz łańcucha (ang. on-chain) i na zewnątrz łańcucha (ang. off-chain), wewnątrzwspólnotowe (ang. in-community) oraz międzywspólnotowe (ang. cross-community) transfery energii, zarządzanie kontraktami krajowymi i zagranicznymi dla transgranicznego rynku pracy.

Rysunek 23 ilustruje konfiguracje inteligentnych kontraktów dla typów transakcji, które mają wspólny zestaw unikalnych obiektów reguł weryfikacji.



Rysunek 23. Konfiguracje korzystają ze wspólnych obiektów reguł weryfikacji z tej samej listy.

Na rysunku (Rysunek 23) zastosowano następujące symbole: obj_i oznacza obiekt i-tej reguły weryfikacji, a ref_i oznacza odniesienie do obj_i. Obiekty reguł weryfikacji są współdzielone w ramach jednego inteligentnego kontraktu, co umożliwia sprawdzanie logicznie powiązanych transakcji. Obiekty reguł weryfikacji są współdzielone na etapie uruchomienia oprogramowania. Dzięki temu wyeliminowano ich redundancję. W efekcie zwiększono efektywność wykorzystania pamięci. Wzorzec jest teraz również dostosowany do obsługi dowolnej liczby typów transakcji.

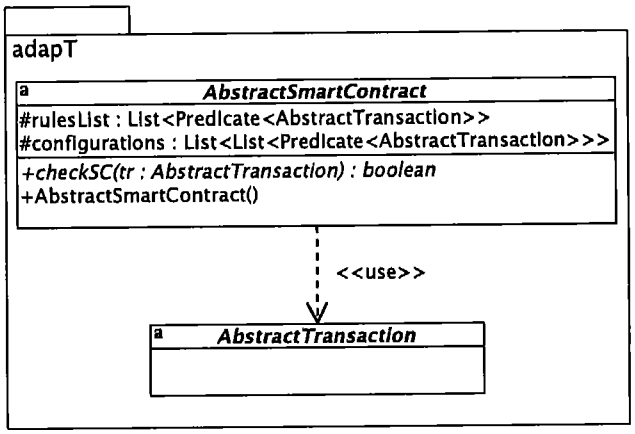
Ponadto autor wprowadził definicję listy obiektów reguły weryfikacji inteligentnego kontraktu.

Definicja 6. Lista obiektów reguł weryfikacji

Lista obiektów reguł weryfikacji to uporządkowana kolekcja obiektów niepowtarzających się reguł weryfikacji dla wszystkich reguł weryfikacji zastosowanych w inteligentnym kontrakcie.

Wzorzec skonstruowano w podziale na dwie warstwy: abstrakcyjną (ang. Abstract) oraz konkretną (ang. Concrete). Dzięki takiemu podziałowi wprowadzono warstwę abstrakcji, wspólną dla wszystkich inteligentnych kontraktów projektowanych według tego schematu. Elementy warstwy abstrakcyjnej są niezależne od realizacji konkretnego inteligentnego kontraktu i są ponownie wykorzystywane w każdym z nich. Warstwa abstrakcyjna składa się z dwóch klas abstrakcyjnych: *AbstractTransaction* i *AbstractSmartContract.*

Rysunek 24 przedstawia diagram klas UML z obiema klasami w warstwie abstrakcyjnej wzorca AdapT.



Rysunek 24. Klasy w warstwie Abstract wzorca AdapT v2.0.

Klasa *AbstractTransaction* służy jako klasa nadrzędna dla wszystkich określonych klas transakcji obsługiwanych przez konkretny inteligentny kontrakt. Wszystkie określone klasy transakcji muszą dziedziczyć z tej klasy abstrakcyjnej. Natomiast, klasa *AbstractSmartContract* stanowi szablon dla wszystkich określonych klas inteligentnych kontraktów. Klasa ta deklaruje listę obiektów reguł weryfikacji (zmienna *rulesList*). Lista ta wykorzystuje interfejs funkcjonalny *Predicate*, który z kolei działa na typie referencyjnym *AbstractSmartContract.* Klasa ta deklaruje także listę konfiguracji reguł weryfikacji dla różnych typów transakcji (zmienna *configurations*). Taka struktura zapewnia dwie własności: obsługę różnych typów referencyjnych dziedziczących z *AbstractTransaction* oraz użycie wyrażeń lambda do odroczonego wykonania. Dodatkowo klasa *AbstractSmartContract* deklaruje metodę *checkSC()*, która służy do weryfikacji inteligentnego kontraktu. Posiada jeden parametr wejściowy, będący referencją do weryfikowanego obiektu transakcji. W zależności od wyniku

weryfikacji metoda ta zwraca wartość logiczną prawda lub fałsz. Połączenie dziedziczenia z programowania obiektowego i wyrażeń lambda z programowania funkcjonalnego pozwala na przetwarzanie różnych typów transakcji w tej jednej metodzie. Ponieważ są to klasy abstrakcyjne, nie można utworzyć instancji żadnej z nich. Obie klasy z warstwy Abstract wzorca zostały zaimplementowane w języku Java.

Kod źródłowy 3 przedstawia implementację klasy *AbstractSmartContract*.

Kod źródłowy 3. Implementacja klasy AbstractSmartContract w języku Java.

```
package adapT;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public abstract class AbstractSmartContract {
  protected List<Predicate<AbstractTransaction>> rulesList = new ArrayList<>();
  protected List<List<Predicate<AbstractTransaction>>> configurations = new ArrayList<>();

  public AbstractSmartContract(){
    configurations.add(new ArrayList<>());
  }
  public boolean checkSC(AbstractTransaction tr){
    boolean correct = false;
      for (Predicate<AbstractTransaction> vR : configurations.get(0)) {
        correct = vR.test(tr);
        if (!correct) break;
    }
    return correct;
  }
}
```

Kod źródłowy 4 przedstawia implementację klasy *AbstractTransaction*.

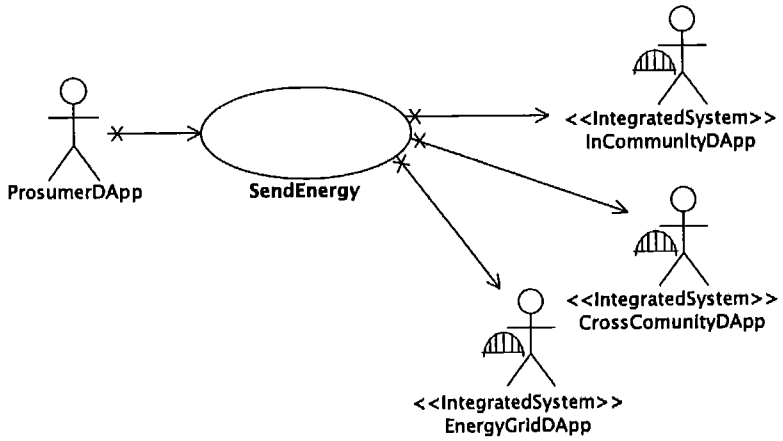
Kod źródłowy 4. Implementacja klasy AbstractTransaction w języku Java

```
package adapT;

public abstract class AbstractTransaction {
}
```

W implementacji wykorzystywane są wyłącznie standardowe klasy i interfejsy dostępne w języku Java. Dodatkowo, kod napisany jest tak, aby był niezależny od domeny, podczas gdy warstwa *Concrete* wzorca wykorzystuje klasy specyficzne dla implementacji konkretnego inteligentnego kontraktu.

Warstwa *Concrete* wzorca wykorzystuje przykład transferu energii pomiędzy różnymi stronami w rozproszonym systemie energii odnawialnej. W takich systemach energia może być wymieniana pomiędzy prosumentami w tej samej społeczności oraz pomiędzy prosumentami w różnych społecznościach. Dodatkowo istnieje możliwość przesłania energii elektrycznej do sieci energetycznej.

Rysunek 25 przedstawia diagram przypadków użycia UML dla przypadku użycia *SendEnergy*. Diagram przypadków użycia wykorzystuje stereotyp <<IntegratedSystem>> z profilu: *UML Profile for Messaging Patterns* [A3].



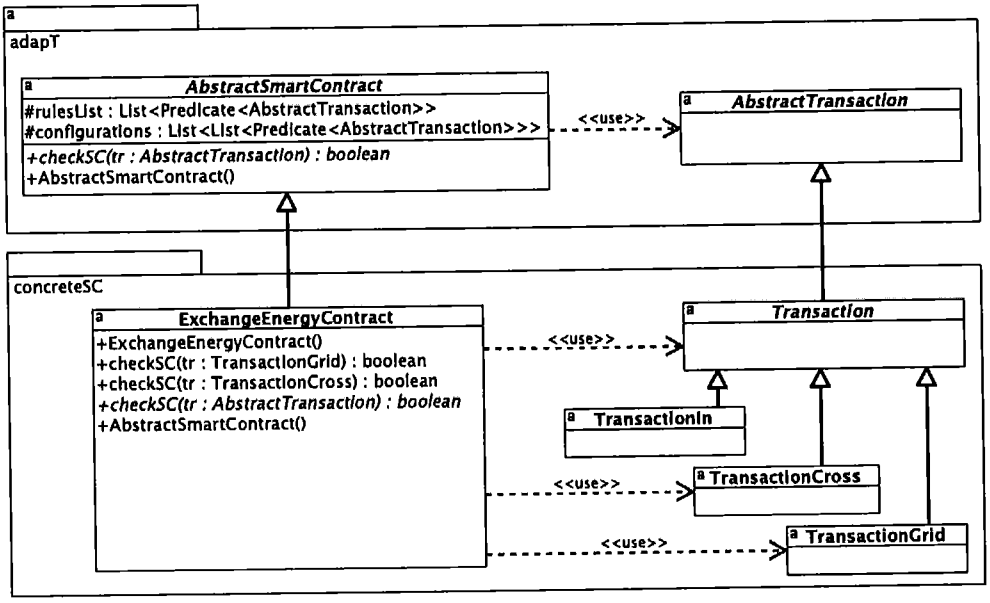
Rysunek 25. Przypadek użycia SendEnergy z różnymi użytkownikami zewnętrznymi.

Działania wykonywane w przypadku użycia są takie same, ale sprawdzane warunki różnią się w zależności od rodzaju transakcji. W przykładzie założono następujący zestaw reguł weryfikacji stosowanych w trzech rozpatrywanych typach transakcji:

- Źródło transakcji musi być inne niż cel transakcji,
- Ilość energii do przesłania musi być większa od zera,
- Nadwyżka energii w węźle źródłowym musi być większa lub równa ilości energii do przesłania,
- Społeczność źródłowa musi różnić się od społeczności docelowej,
- Docelowe zapotrzebowanie musi być większe lub równe ilości energii do przesłania,
- Celem jest podsieć sieci energetycznej.

Korzystając ze wzorca AdapT, przypadek użycia *SendEnergy* można zaimplementować za pomocą jednego inteligentnego kontraktu.

Rysunek 1 przedstawia diagram klas UML dla przypadku użycia *SendEnergy*. Przedstawiono zarówno klasy abstrakcyjne jak i zaimplementowane klasy konkretnego inteligentnego kontraktu *ExchangeEnergy*.



Rysunek 26. Wzorzec AdapT z klasami abstrakcyjnymi i konkretnymi.

W przedstawionym przykładzie klasa *ExchangeEnergyContract* dziedziczy po klasie abstrakcyjnej *AbstractSmartContract*. W konstruktorze tej klasy inteligentnego kontraktu inicjowana jest zarówno lista reguł weryfikacji, jak i konfiguracje typów transakcji.

Kod źródłowy 5 przedstawia implementację konstruktora klasy *ExchangeEnergyContract* w języku Java.

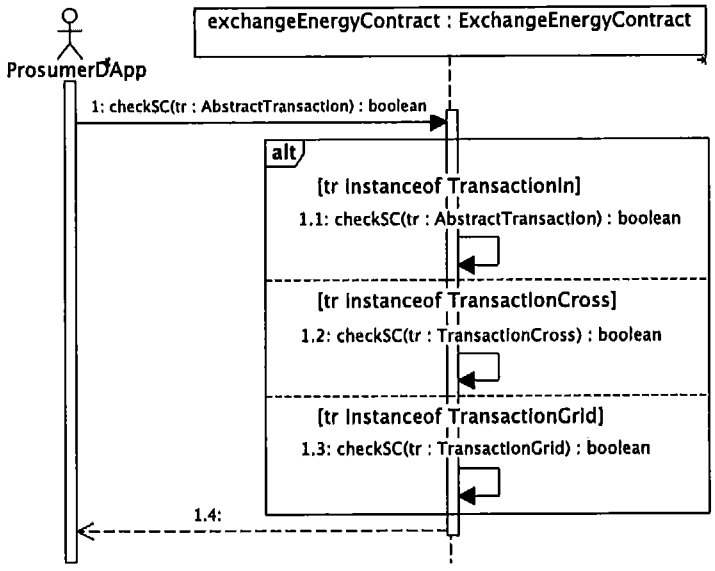
Kod źródłowy 5. Implementacja konstruktora klasy ExchangeEnergyContract.

```
public ExchangeEnergyContract(){
 // verification rules
 rulesList.add(t -> ((Transaction) t).getSourceID() != ((Transaction) t).getTargetID());
 rulesList.add(t -> ((Transaction) t).getQuantity() > 0);
 rulesList.add(t -> ((Transaction) t).getSourceSurplus() >= ((Transaction) t).getQuantity());
 rulesList.add(t -> ((TransactionCross) t).getSourceCommunityID() != ((TransactionCross) t).getTargetCommunityID());
 rulesList.add(t -> ((Transaction) t).getTargetNeed() >= ((Transaction) t).getQuantity());
 rulesList.add(t -> ((TransactionGrid) t).getTargetID() == ((TransactionGrid) t).getEnergySubnetID());
 // configurations
 for (int i = 1; i <= 2 ; i++) configurations.add(new ArrayList<>());
 // configure rules for TransactionIn
 configurations.get(0).add(rulesList.get(0));
 configurations.get(0).add(rulesList.get(1));
 configurations.get(0).add(rulesList.get(2));
 // configure rules for TransactionGrid
 configurations.get(1).add(rulesList.get(0));
 configurations.get(1).add(rulesList.get(1));
 configurations.get(1).add(rulesList.get(2));
 configurations.get(1).add(rulesList.get(5));
 // configure rules for TransactionCross
```

```
configurations.get(2).add(rulesList.get(3));
configurations.get(2).add(rulesList.get(0));
configurations.get(2).add(rulesList.get(1));
configurations.get(2).add(rulesList.get(2));
configurations.get(2).add(rulesList.get(4));
}
```

Metoda *checkSC()* zaimplementowana w klasie abstrakcyjnej inteligentnego kontraktu działa na pierwszej konfiguracji reguł weryfikacji. W konkretnej klasie inteligentnego kontraktu metoda ta powinna być przeciążana tyle razy, ile jest dodatkowych typów transakcji. W rozważanym przykładzie należało napisać jeszcze dwie metody. Jedna metoda dla typu transakcji między społecznościami i jedna dla typu transakcji do sieci. Istotnym jest, że jeśli metoda *checkSC()* zostanie wywołana z typem transakcji innym niż zadeklarowany dla inteligentnego kontraktu, zostanie wykonana poprawnie i zwróci wartość logiczną *fałsz*. Wartość logiczna *prawda*, świadcząca o poprawności weryfikacji transakcji, może zostać zwrócona tylko dla jednego z rozpatrywanych typów transakcji.

Podczas uruchomienia metody *checkSC()* Java weryfikuje typ parametru i wywołuje odpowiednią metodę spośród przeciążonych. Wywołanie przeciążonej metody checkSC() przedstawiono na diagramie sekwencji UML (Rysunek 27).



Rysunek 27. Wywołanie metody weryfikacji inteligentnego kontraktu.

Efektywność wykorzystania kodu źródłowego oprogramowania wpływa na jego utrzymanie. Podniesienie poziomu jego ponownego użycia ułatwia modyfikacje i zmniejsza zakres testowania. Autor wprowadził miarę U^{SC} jako procent ponownie wykorzystanych reguł weryfikacji w ramach inteligentnego kontraktu.

Miarę U^{sc} wyrażono następującym równaniem:

$$U^{sc} = \frac{\sum_{i=1}^{C}(\frac{u_i^r}{u_i})}{C} \times 100$$

gdzie:

C – liczba konfiguracji w inteligentnym kontrakcie,

u_i^r – liczba ponownie użytych reguł weryfikacji w i-tej konfiguracji inteligentnego kontraktu,

u_i – liczba reguł weryfikacji w i-tej konfiguracji inteligentnego kontraktu.

Obliczono procent ponownie wykorzystanych reguł weryfikacji dla rozpatrywanego w artykule inteligentnego kontraktu: U^{sc} = 58,3 %.

Do obliczeń wzięto konfiguracje w kolejności od najliczniejszej do najmniej licznej. Wynik powyżej 50 % wskazuje na wysoki poziom ponownego wykorzystania kodu źródłowego reguł weryfikacji. Należy pamiętać, że gdyby nie zastosowano wzorca projektowego, taki byłby poziom redundancji reguł weryfikacji.

W czasie wykonywania warto także określić poziom efektywności wykorzystania zestawu obiektów reguł weryfikacji w konfiguracjach. Autor zaproponował miarę D^{sc} jako procent redundantnych obiektów reguł weryfikacji dla inteligentnego kontraktu. Miarę wyrażono następującym równaniem:

$$D^{sc} = \frac{(\sum_{i=1}^{C} o_i^c) - v^{sc}}{v^{sc}} \times 100$$

gdzie:

o_i^c – liczba nowo utworzonych obiektów w i-tej konfiguracji inteligentnego kontraktu,

v^{sc} – liczba unikalnych reguł weryfikacji w inteligentnym kontrakcie.

Obliczono wartość procentową nadmiarowych obiektów reguł weryfikacji D^{sc} dla rozpatrywanego w artykule smart kontraktu:

$$D^{sc} = \frac{(3+1+2)-6}{6} \times 100 = 0\ \%$$

Oznacza to, że obiekty reguł weryfikacji są w pełni wykorzystywane przez konfiguracje. Ponadto dokonanie zmian w sprawdzaniu typów transakcji nie powoduje tworzenia nowych obiektów ani usuwania istniejących. W rezultacie Garbage Collector nie jest angażowany w trakcie działania inteligentnego kontraktu. Należy także podkreślić, że żaden obiekt reguły weryfikacji z listy obiektów reguły weryfikacji nie pozostaje niewykorzystany.

W niedawno opublikowanym artykule badawczym Khan i in. [27] wykazali, że ogólny współczynnik klonowania inteligentnych kontraktów Solidity wynosi 30,13%, z czego 27,03% to dokładne duplikaty. Zastosowanie wzorca AdapT do projektowania inteligentnego kontraktu

pozwala na osiągnięcie wartości 0% współczynnika redundantnych reguł weryfikacji. Dlatego opracowanie wzorców projektowych zwiększających poziom ponownego wykorzystania kodu źródłowego inteligentnych kontraktów wydaje się jednym z właściwych kierunków prac badawczych.

Celem testów wydajnościowych było sprawdzenie, jak szybko transakcje są sprawdzane przez inteligentny kontrakt zaprojektowany według wzorca AdapT. Jeśli chodzi o środowisko wykonawcze, testy przeprowadzono na MacBooku Air z procesorem Apple M2, 16 GB pamięci RAM i dysku Solid State Drive (SSD) 256 GB. MacBook Air działał pod systemem operacyjnym macOS Sonoma 14.3. Aby przeprowadzić testy wydajnościowe, zaprojektowano osobną klasę testowania inteligentnych kontraktów *TestContract*. Klasa ta zawiera dwie metody: *conductTest()* i *runTests()*. Pierwsza metoda mierzy czas ewaluacji inteligentnego kontraktu. W metodzie tej wykorzystano metodę systemową *System.nanoTime()* w celu uzyskania pomiaru czasu z dokładnością do jednej nanosekundy. Druga metoda odpowiada za wykonanie odpowiedniej liczby powtórzeń pomiaru.

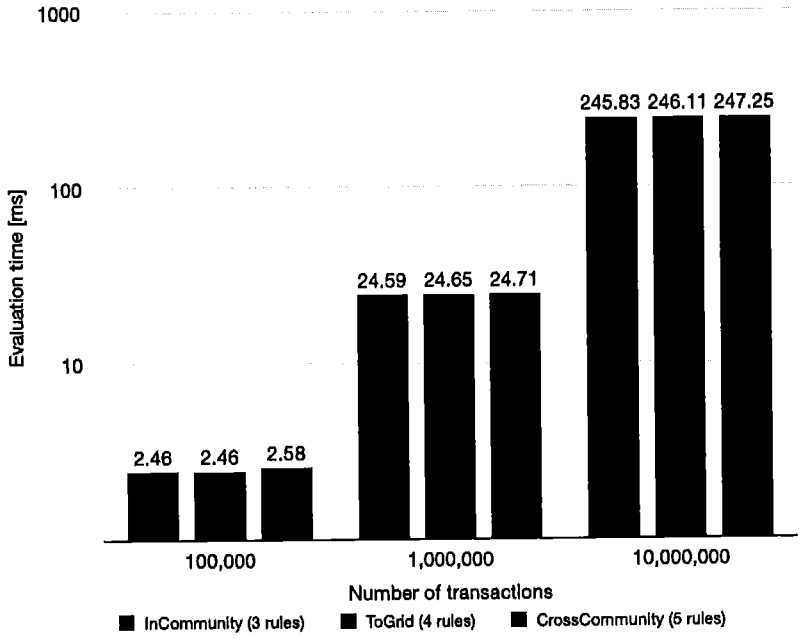
Kod źródłowy 6 przedstawia implementację klasy *TestContract*.

Kod źródłowy 6. Implementacja klasy TestContract do testów wydajnościowych inteligentnego kontraktu.

```
package tests;

import adapT.AbstractSmartContract;
import adapT.AbstractTransaction;

public class TestContract {
   private static int[] numberTransactions = new int[]{100000, 1000000, 10000000};
   private static long conductTest(AbstractSmartContract sC, AbstractTransaction tR, int numberTransactions, int
repetitions){
      boolean correct = false;
      long sum = 0;
      long startTime = 0, endTime = 0;
      for (int j = 0; j <= repetitions; j++){
         startTime = System.nanoTime();
         for (int i = 0; i < numberTransactions; i++) correct = sC.checkSC(tR);
         endTime = System.nanoTime();
         if( j > 0 ) sum = sum + (endTime - startTime);
      }
      return sum/repetitions;
   }

   public static void runTests(AbstractSmartContract sC, AbstractTransaction tR, int repetitions){
      long executionTime = 0;
```

```
    // smart contract evaluation time
    System.out.println(``Evaluation time of transaction: " + tR.getClass());
    for (int numberTransaction : numberTransactions) {
      System.out.println(``Number of processed transactions: " + numberTransaction);
      executionTime = conductTest(sC, tR, numberTransaction, repetitions);
      System.out.println(``Execution time: " + executionTime);
    }
  }
}
```

Jako podstawową miarę efektywności przyjęto czas oceny określonej liczby transakcji przez inteligentny kontrakt E_n^{sc}. Każdy test powtórzono 50 razy. Testy przeprowadzono dla inteligentnego kontraktów z 3, 4 i 5 regułami weryfikacji. W sumie przeprowadzono 450 testów. Rysunek 28 przedstawia wyniki testu dla liczby transakcji w następującym zakresie, <100 000; 10 000 000>. Wyniki na rysunku przedstawiono w skali logarytmicznej.

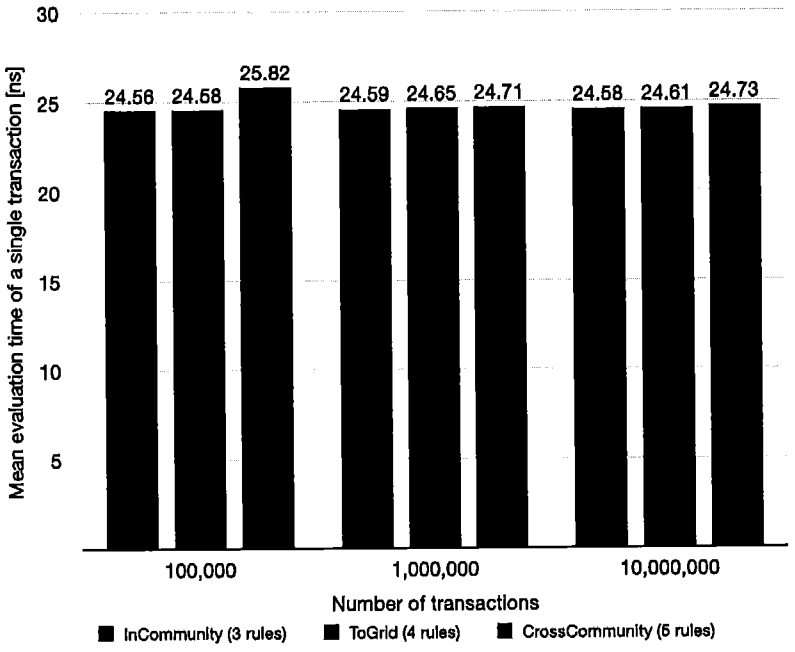


Rysunek 28. Czas ewaluacji liczby transakcji w zakresie <100 000; 10 000 000> przez inteligentny kontrakt.

Na podkreślenie zasługuje jeden z faktów wynikających z wyników przedstawionych na Rysunek 28. Czas oceny 10 000 000 transakcji wynosi poniżej 0,25 s, niezależnie od uwzględnianej liczby reguł weryfikacji w inteligentnym kontrakcie. Obecnie środowisko łańcucha bloków Solana uważane jest za jedno z najszybszych oferujący przepustowość do kilkudziesięciu tysięcy transakcji na sekundę [28]. Uzyskane wyniki ilustrują potencjał wydajności inteligentnych kontraktów zaprojektowanych według wzorca AdapT.

Warto też zwizualizować średni czas ewaluacji pojedynczej transakcji. Rysunek 29 przedstawia wartości tej miary obliczone na podstawie wyników uzyskanych przy ocenie

rozpatrywanych wolumenów transakcji. Wyniki przedstawiono w skali liniowej i wyrażono w nanosekundach.



Rysunek 29. Średni czas ewaluacji pojedynczej transakcji przez inteligentny kontrakt.

Średni czas ewaluacji pojedynczej transakcji przez inteligentny kontrakt, jest praktycznie stały i wynosi około 25 nanosekund. Oznacza to, że mechanizm walidacji inteligentnych kontraktów we wzorcu został zaprojektowany poprawnie. Wydłużenie czasu oceny jest wprost proporcjonalne do liczby ocenionych transakcji.

Wyniki testów wydajności wyraźnie pokazują, że inteligentne kontrakty jako oprogramowanie mają ogromny potencjał do przetwarzania dużych wolumenów transakcji, znacznie przekraczający możliwości obecnie dostępnych środowisk.

IV.4.8. Metoda redukcji zestawu testów dla inteligentnego kontraktu

Kolejnym istotnym elementem cyklu publikacji jest opracowana metoda k+1 redukcji zestawu testów. Metoda umiejscowiona jest w widoku architektonicznych *Kontraktów*. Opracowany wzorzec projektowania inteligentnych kontraktów umożliwia redukcję liczby przypadków testowych. Zdefiniowano metodę k+1 testowania inteligentnych kontraktów.

Habilitant przedstawił założenia metody k+1 w opublikowanym komunikacie [A5].

Istotą zaproponowanej metody k+1 jest podejście proaktywne. Polega na analizie mechanizmu przetwarzania reguł weryfikacji na etapie tworzenia oprogramowania. Zamiast powtarzać dziesiątki, setki czy tysiące testów, metoda skupia się na odpowiedniej konstrukcji minimalnego zestawu testów zapewniających pełne pokrycie weryfikowanych warunków. Poza

tym metoda minimalizuje również koszty budowania, utrzymywania i uruchamiania testów. Osiąga się to poprzez minimalizację liczby przypadków testowych i zastosowanie nieskomplikowanych warunków logicznych dla pojedynczych testów.

Na inteligentny kontrakt mogą być nałożone różne reguły weryfikacji. Natomiast, wszystkie one muszą być spełnione, aby wyrażenie ewaluacji mogło przyjąć wartość logiczną *prawda*. Wartość wyrażenia ewaluacji E^{val} można zatem wyrazić za pomocą iloczynu logicznego wartości poszczególnych reguł weryfikacji:

$$E^{val} = \prod_{i=1}^{k} v_i$$

gdzie:

k – liczba reguł weryfikacji w wyrażeniu ewaluacji,

v_i – wartość ewaluacji i-tej reguły weryfikacji.

Pozytywna weryfikacja inteligentnego kontraktu następuje tylko wtedy, gdy każda z zawartych w nim reguł weryfikacji zwróci wartość prawdziwą. W rezultacie jeden przypadek testowy weryfikuje pozytywną ocenę inteligentnego kontraktu. Pojedyncza reguła weryfikacji może indywidualnie prowadzić do tego, że wyrażenie ewaluacji zwróci fałszywą wartość. W przypadku testowym tej reguły, wszystkie reguły weryfikacji muszą zwrócić wartość prawdziwą, z wyjątkiem tej która jest weryfikowana.

Aby zilustrować opisaną regułę, w Tabela 16 przedstawiono zestaw przypadków testowych dla ośmiu reguł weryfikacji. Dla każdego przypadku testowego (PT) pokazane są wartości logiczne wszystkich reguł weryfikacji. Liczba całkowita 1 oznacza wartość prawdziwą, a 0 oznacza wartość fałszywą.

Tabela 16. Przypadki testowe w metodzie k+1 dla inteligentnego kontraktu zawierającego 8 reguł weryfikacji.

PT	Wynik	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
1	Prawda	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
2	Fałsz	**0**	1	1	1	1	1	1	1
3	Fałsz	1	**0**	1	1	1	1	1	1
4	Fałsz	1	1	**0**	1	1	1	1	1
5	Fałsz	1	1	1	**0**	1	1	1	1
6	Fałsz	1	1	1	1	**0**	1	1	1
7	Fałsz	1	1	1	1	1	**0**	1	1
8	Fałsz	1	1	1	1	1	1	**0**	1
9	Fałsz	1	1	1	1	1	1	1	**0**

Dlatego liczba przypadków testowych wymaganych do przetestowania działania inteligentnego kontraktu jest większa niż liczba reguł weryfikacji o jeden. Zatem, liczbę przypadków testowych w metodzie T^M można wyrazić równaniem:

$$T^M = k + 1$$

Dla porównania, liczbę przypadków testowych dla pełnego pokrycia T^{FC} można wyznaczyć za pomocą wariacji z powtórzeniami i wyrazić równaniem:

$$T^{FC} = \bar{V}_n^k = n^k$$

gdzie: n – liczba elementów w zbiorze wartości N = {0, 1}.

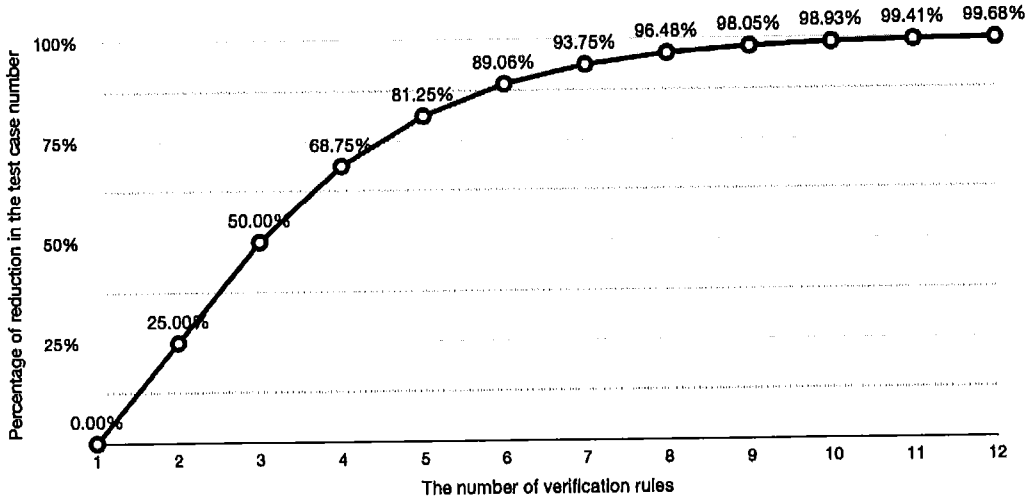
Dla ośmiu reguł weryfikacji mamy następujące wartości zmiennych: n = 2, k = 8. W rezultacie liczba przypadków testowych dla pełnego pokrycia wynosi $T^{FC} = 2^8 = 256$, natomiast dla metody k+1 wynosi $T^M = 8 + 1 = 9$.

Procentową redukcję liczby przypadków testowych można wyrazić równaniem:

$$R_{sc} = \frac{T^{FC} - T^M}{T^{FC}} \times 100 = \frac{n^k - (k+1)}{n^k} \times 100$$

Przy ośmiu regułach weryfikacji (k = 8) metoda k+1 w stosunku do pełnego pokrycia redukuje liczbę przypadków testowych o 96,48 %. Wraz ze zwiększaniem liczby reguł weryfikacji w wyrażeniu ewaluacji następuje wzrost liczby redukowanych przypadków testowych. Przy jedenastu regułach weryfikacji (k = 11) redukowanych jest 99,41 % przypadków testowych.

Rysunek 30 przedstawia zależność procentowej redukcji liczby przypadków testowych w zależności od liczby reguł weryfikacji. Począwszy od siedmiu reguł weryfikacji w wyrażeniu ewaluacji w inteligentnym kontrakcie, redukcja liczby przypadków testowych przekracza 90%.



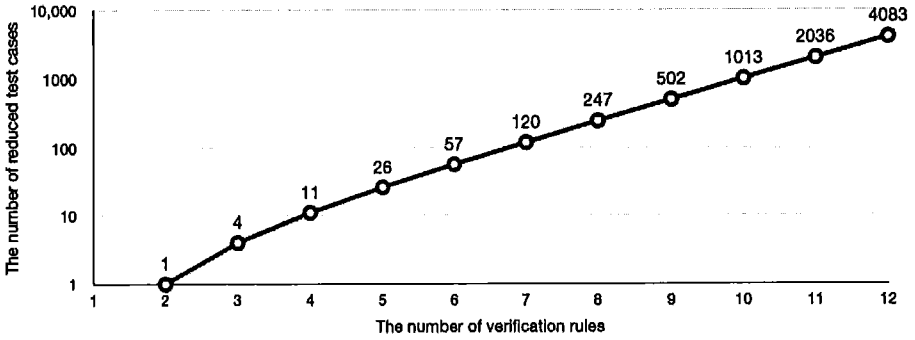
Rysunek 30. Procent redukcji liczby przypadków testowych w zależności od liczby reguł weryfikacji.

Tabela 17 zawiera dane umożliwiające porównanie liczby przypadków testowych dla pełnego pokrycia z liczbą przypadków testowych przy zastosowanym metody k+1.

Tabela 17. Porównanie liczby przypadków testowych dla metody k+1 i pełnego pokrycia.

Liczba reguł	Liczba PT przy pełnym pokryciu	Liczba PT przy metodzie k+1	Liczba zredukowanych PT	R_{sc}
1	2	2	0	0,00 %
2	4	3	1	25,00 %
3	8	4	4	50,00 %
4	16	5	11	68,75 %
5	32	6	26	81,25 %
6	64	7	57	89,06 %
7	128	8	120	93,75 %
8	256	9	247	96,48 %
9	512	10	502	98,05 %
10	1024	11	1013	98,93 %
11	2048	12	2036	99,41 %
12	4196	13	4083	99,68 %

Rysunek 31 przedstawia wykres zredukowanej liczby przypadków testowych dla różnej liczby reguł weryfikacji w wyrażeniu ewaluacji (zastosowano skalę logarytmiczną). Im więcej jest reguł weryfikacji w wyrażeniu ewaluacji, tym większa oszczędność w liczbie zredukowanych przypadków testowych.



Rysunek 31. Liczba zredukowanych przypadków testowych w zależności od liczby reguł weryfikacji.

Od liczby 10 reguł weryfikacji oszczędności w liczbie przypadków testowych wyrażają się już w tysiącach. Zmniejszenie liczby przypadków testowych ma pozytywny wpływ na czas przygotowania zestawu testów. Przygotowanie tak okrojonego zestawu testów jest także mniej podatne na błędy. Utrzymanie zdecydowanie mniejszej liczby przypadków testowych jest też łatwiejsze i mniej kosztowne.

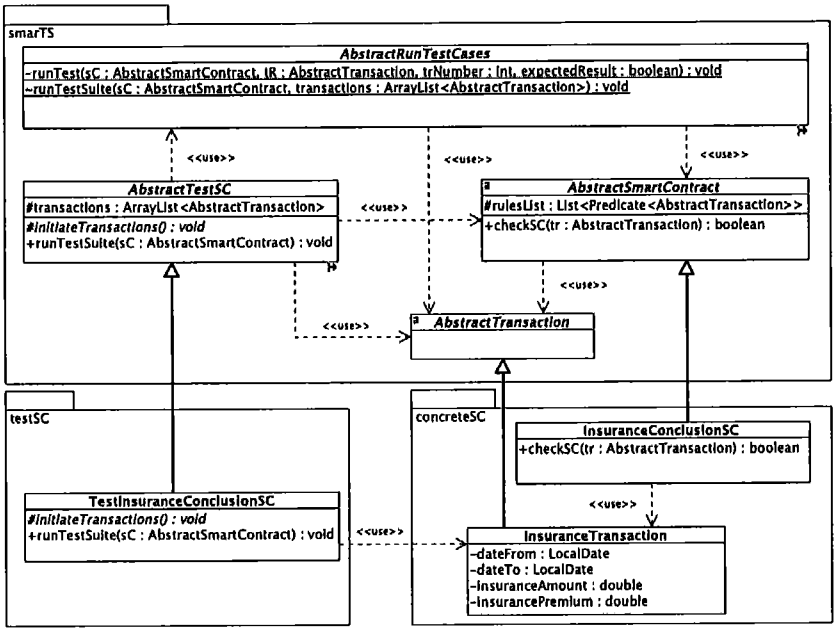
Ponadto, habilitant oprogramował pakiet *SmarTS* do generowania i wykonywania zestawu testów dla inteligentnego kontraktu zgodnie z metodą k+1. Pakiet ten habilitant opublikował w postaci artykułu typu *Original software publication* [A1]. Pakiet SmarTS został zaimplementowany w języku programowania Java, a jego kod źródłowy został udostępniony w publicznych repozytoriach [29], [30].

Struktura pakietu zawiera warstwę abstrakcji, która jest komponentem wielokrotnego użytku. Komponent ten można bezpośrednio wykorzystać w projektach rozwoju inteligentnych kontraktów. Taki sposób projektowania jest zgodny z najnowszymi kierunkami w konstruowaniu oprogramowania łatwego w utrzymaniu [31].

Warstwa abstrakcyjna pakietu SmartTS składa się z czterech klas:

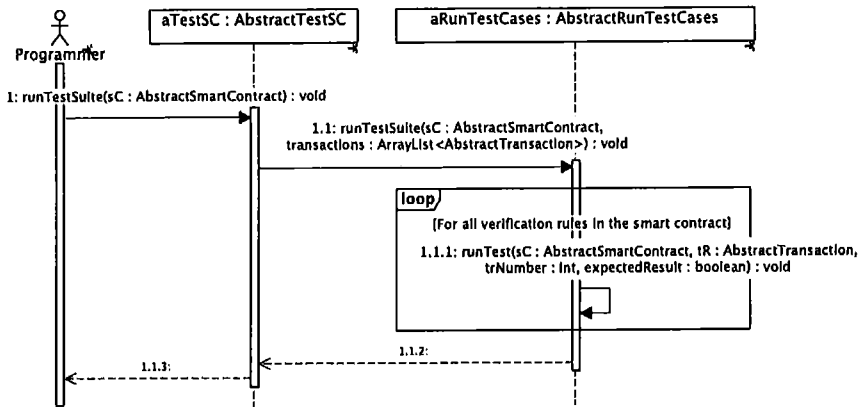
- klasa AbstractTransaction – klasa abstrakcyjna reprezentująca ogólną transakcję.
- klasa AbstractSmartContract – klasa abstrakcyjna definiująca ogólny inteligentny kontrakt. Klasa deklaruje strukturę danych do przechowywania reguł weryfikacji inteligentnych kontraktów. Reguła pojedynczej weryfikacji jest deklarowana jako typ Predicate<AbstractTransaction>. Implementację mechanizmu weryfikacji transakcji zapewniono w metodzie checkSC(). Metodę oznaczono jako ostateczną (final), co uniemożliwia jej przesłonięcie.
- klasa AbstractTestSC – jest to jedyna klasa publiczna w pakiecie. Klasa udostępnia jedną metodę jako publiczną, runTestSuite(). Metoda jest zaimplementowana w klasie abstrakcyjnej i oznaczona opcjonalnym modyfikatorem final, który blokuje możliwość przesłaniania metody runTestSuite() w rzeczywistych klasach testowych. W ten sposób wymuszono jednolity mechanizm działania metody dla wszystkich rzeczywistych klas testowych inteligentnych kontraktów. Z drugiej strony metoda initiateTransactions() jest abstrakcyjna i musi zostać zaimplementowana przez określoną klasę testową inteligentnego kontraktu. Metoda powinna tworzyć obiekty transakcyjne potrzebne do wykonania zestawu testów dla inteligentnego kontraktu.
- klasa AbstractRunTestCases – modyfikator dostępu pakiet-prywatny zastosowany do tej klasy ogranicza jej widoczność i dostępność tylko w pakiecie. Sama klasa abstrakcyjna udostępnia tylko runTestSuite(). Ogranicza jednak widoczność metody, stosując chroniony modyfikator dostępu, tylko do klas w tym samym pakiecie lub klas podrzędnych. Za wykonanie pojedynczego przypadku testowego odpowiedzialna jest metoda prywatna runTest().

Kompletny pakiet oprogramowania SmarTS zawiera trzy pakiety języka Java. Oprócz pakietu *smarTS* w rozwiązaniu dla testowania konkretnego inteligentnego kontraktu występują jeszcze dwa pakiety. Pakiet *concreteSC* zawiera klasy inteligentnego kontraktu oraz transakcji. Natomiast, pakiet *testSC* zawiera klasę testową dla konkretnego inteligentnego kontraktu Rysunek 32 przedstawia diagram klas UML z klasami abstrakcyjnymi w pakiecie SmarTS, a także przykładowymi konkretnymi klasami inteligentnego kontraktu, transakcji i testową.



Rysunek 32. Klasy w pakiecie oprogramowania SmarTS.

Główną funkcją oferowaną przez pakiet SmarTS jest generowanie i uruchamianie kolekcji zestawu testów dla wszystkich zaimplementowanych w rozwiązaniu inteligentnych kontraktów. Opracowana biblioteka udostępnia publiczną metodę runTestSuite() w klasie AbstractTestSC umożliwiającą uruchomienie zestawu testów dla pojedynczego inteligentnego kontraktu. Rysunek 33 przedstawia diagram sekwencji UML dla wywołania metody runTestSuite().



Rysunek 33. Kolejność wywoływania metod podczas uruchamiania zestawu testów.

Warto zaznaczyć, że metoda działa na typach abstrakcyjnych zarówno dla inteligentnego kontraktu, jak i przetwarzanej transakcji. Przykład uruchomienia wielu zestawów testów dla dwóch różnych inteligentnych kontraktów został zaimplementowany w klasie RunTestSuite, również zawartej w udostępnionych repozytoriach [29], [30].

Działanie klas abstrakcyjnych jest niezależne od testowanych prawdziwych inteligentnych kontraktów i sprawdzanych przez nie transakcji. Poniżej przedstawiono kody źródłowe klas pakietu *smarTS*, które generują i wykonują przypadki testowe.

Kod źródłowy 1 przedstawia implementację klasy AbstractTestSC.

Kod źródłowy 7. Klasa AbstractTestSC.

```
package pl.gdansk.ug.smarTS;

import java.util.ArrayList;
public abstract class AbstractTestSC {
 protected ArrayList<AbstractTransaction> transactions = new ArrayList<>();
 public AbstractTestSC(){
     initiateTransactions();
  }
 protected abstract void initiateTransactions();
 public final void runTestSuite(AbstractSmartContract sC){
     AbstractRunTestCases.runTestSuite(sC, transactions);
  }
}
```

Z klasy AbstractTestSC musi dziedziczyć konkretna klasa testowa. Metodę initiateTransactions() zdefiniowano jako abstrakcyjną metodę instancji, aby wymusić jej przesłonięcie w klasie potomnej.

Natomiast, klasa AbstractRunTestCases jest klasą użytkową zawierającą wyłącznie metody klasy. Obie metody w tej klasie są oznaczone słowem kluczowym static i wywoływane w kontekście klasy. Wszelkie dane potrzebne do wykonania tych metod przekazywane są do tych metod w postaci parametrów wejściowych. Wszystkie wywołania metod są wykonywane na typie referencyjnym. Klasa może działać dla dowolnego kontraktu dziedziczącego z klasy AbstractSmartContract. Metoda runTestSuite() uruchamia przypadki testowe dla konkretnego inteligentnego kontraktu. Metoda wykorzystuje listę obiektów transakcji zadeklarowanych w konkretnym inteligentnym kontrakcie. Iteruje całą listę transakcji testowych i wywołuje metodę runTest() dla każdej transakcji. Metoda runTest() wykonuje przypadek testowy. Ta metoda z kolei wywołuje metodę checkSC() w konkretnym inteligentnym kontrakcie, która sprawdza wartość zwróconą przez wyrażenie oceny.

Kod źródłowy przedstawia implementację klasy AbstractRunTestCases.

Kod źródłowy 8. Klasa AbstractRunTestCases.

```
package pl.gdansk.ug.smarTS;

import java.util.ArrayList;
abstract class AbstractRunTestCases {
 private static void runTest(AbstractSmartContract sC, AbstractTransaction tR, int trNumber, boolean expectedResult){
  boolean result = sC.checkSC(tR);
  boolean correct = result == expectedResult;
  System.out.println("Test no: " + (trNumber + 1) + ", test result: " + ((correct)?"PASS":"FAIL"));
 }
 static void runTestSuite(AbstractSmartContract sC, ArrayList<AbstractTransaction> transactions){
  System.out.println("Smart contract class: " + sC.getClass());
    for (int i = 0; i < transactions.size(); i++){
      runTest(sC, transactions.get(i),i, i == 0);
  }
 }
}
```

Obydwie te klasy testowe (AbstractTestSC, AbstractRunTestCases) działają na klasach abstrakcyjnych AbstractSmartContract i AbstractTransaction, reprezentujących odpowiednio ogólną formę inteligentnego kontraktu i transakcji.

Aby zademonstrować funkcjonalność pakietu, przygotowano dwa inteligentne kontrakty z zupełnie różnych domen. Pierwszy kontrakt zawiera reguły weryfikujące realizację przesłania energii pomiędzy prosumentami odnawialnych źródeł energii. Ten inteligentny kontrakt SendEnergySC zawiera osiem reguł weryfikacji. Drugi inteligentny kontrakt InsuranceConclusionSC odpowiada za weryfikację danych transakcji ubezpieczeniowych. Składa się na niego pięć reguł weryfikacji.

Kod źródłowy 9 przedstawia implementację klasy InsuranceConclusionSC. Indywidualne reguły weryfikacji definiowane są za pomocą wyrażeń lambda.

Kod źródłowy 9. Klasa InsuranceConclusionSC.

```
package pl.gdansk.ug.concreteSC;

import pl.gdansk.ug.smarTS.AbstractSmartContract;
import java.time.LocalDate;
import java.util.Arrays;

public class InsuranceConclusionSC extends AbstractSmartContract {
 public InsuranceConclusionSC(){
 rulesList = Arrays.asList(
  t -> !((InsuranceTransaction) t).getDateFrom().isBefore(LocalDate.now()),
```

```
 t -> ((InsuranceTransaction) t).getDateFrom().isBefore(((InsuranceTransaction) t).getDateTo()),
 t -> ((InsuranceTransaction) t).getInsuranceAmount() > 0,
 t -> ((InsuranceTransaction) t).getInsurancePremium() > 0,
 t -> ((InsuranceTransaction) t).getInsurancePremium() < ((InsuranceTransaction) t).getInsuranceAmount());
 }
}
```

Klasa InsuranceConclusionSC dziedziczy metodę checkSC() z klasy AbstractSmartContract. Metoda ta sprawdza, czy zostały spełnione warunki umożliwiające sfinalizowanie transakcji. Posiada jeden parametr wejściowy, będący referencją do weryfikowanego obiektu transakcji. Implementację metody można uznać za czystą (ang. pure), ponieważ wynik zależy wyłącznie od danych wejściowych. W zależności od wyniku weryfikacji metoda zwraca wartość logiczną *prawda* lub *fałsz*. Połączenie dziedziczenia z programowania obiektowego i wyrażeń lambda z programowania funkcjonalnego pozwala tą jedną metodą sprawdzać różne rodzaje transakcji. Metodę zaimplementowano jako ostateczną (ang. final). Zatem metoda działa tak samo dla wszystkich potomnych inteligentnych kontraktów.

Klasa TestInsuranceConclusionSC przesłania (ang. override) metodę initiateTransactions() klasy abstrakcyjnej AbstractTestSC. Zgodnie ze wzorcem testowym k+1, dla 5 reguł weryfikacji w inteligentnym kontrakcie InsuranceConclusionSC, tworzonych jest 6 obiektów transakcji w klasie TestInsuranceConclusionSC. Obie te klasy operują na obiektach klasy InsuranceTransaction.

Kod źródłowy 10 przedstawia implementację klasy TestInsuranceConclusionSC.

Kod źródłowy 10. Klasa TestInsuranceConclusionSC.

```
package pl.gdansk.ug.testSC;

import pl.gdansk.ug.concreteSC.InsuranceTransaction;
import pl.gdansk.ug.smarTS.AbstractTestSC;
import java.time.LocalDate;

public class TestInsuranceConclusionSC extends AbstractTestSC {
 public void initiateTransactions(){
 transactions.clear();
 transactions.add(new InsuranceTransaction(LocalDate.now(), LocalDate.now().plusYears(1),250.0,10000.0));
 transactions.add(new InsuranceTransaction(LocalDate.now().minusDays(1), LocalDate.now().plusYears(1),250.0,10000.0));
 transactions.add(new InsuranceTransaction(LocalDate.now(), LocalDate.now().minusMonths(1),250.0,10000.0));
 transactions.add(new InsuranceTransaction(LocalDate.now(), LocalDate.now().plusYears(1),250.0,0.0));
 transactions.add(new InsuranceTransaction(LocalDate.now(), LocalDate.now().plusYears(1),0.0,10000.0));
 transactions.add(new InsuranceTransaction(LocalDate.now(), LocalDate.now().plusYears(1),250.0,100.0));
 }
}
```

Zestawy testów dla obu inteligentnych kontraktów można wykonać, uruchamiając klasę RunTestSuite. W klasie RunTestSuite zadeklarowano dwie zmienne klasy. Zmienna smartContractList typu List<AbstractSmartContract> przechowuje obiekty klas potomnych klasy AbstractSmartContract, czyli konkretne inteligentne kontrakty. Ponadto zmienna testClassesList typu List<AbstractTestSC> przechowuje obiekty klas potomnych klasy AbstractTestSC, czyli konkretnych klas testowych. Programista musi dodać obiekty obu kategorii do zadeklarowanych zmiennych, implementując dwie metody: createSmartContracts() i createTestClasses(). Uruchamianie zestawów testów dla wszystkich inteligentnych kontraktów przechowywanych w zmiennej smartContractList jest zaimplementowane w metodzie main(). Metoda uruchamia metody createSmartContracts() oraz createTestClasses(). Następnie w pętli for iteruje po wszystkich inteligentnych kontraktach w kolekcji ArrayList. Pakiet SmarTS generuje i wykonuje zestaw testów dla inteligentnego kontraktu w czasie krótszym niż 0,01 milisekundy. Wyniki uzyskano dla inteligentnych kontraktów z maksymalnie ośmioma regułami weryfikacji. Warto podkreślić, że mechanizm generowania i uruchamiania przypadków testowych został zaprojektowany i wdrożony tak, aby był niezależny od konkretnych inteligentnych kontraktów. Dzięki temu pakiet można wykorzystać do inteligentnych kontraktów z dowolnej domeny biznesowej.

IV.4.9. Zastosowania modelu widoków architektonicznych 1+5

Potrzeba konstrukcji takiego podejścia wyniknęła z doświadczeń w analizie oraz projektowaniu systemów informatycznych (Straż Graniczna RP, PKP CARGO, Zone Vision Ltd.). W każdym z tych realizowanych projektów zachodziła potrzeba komunikacji praktycznie w trybie rzeczywistym z zewnętrznymi systemami informatycznymi. W przypadku systemu Odprawa dla Straży Granicznej RP były to m.in. systemy: SISOne4All (Portugalia), CEWiUDO, CEPiK, Ewidencja zleceń (Straż Graniczna RP). Współpraca między systemami często realizowana była i jest w postaci wywoływania usług. Na komunikację między systemami nakładane są wymagania jakościowe w postaci umów poziomu dostarczanych usług (ang. Service Level Agreement). Poszczególne funkcje systemu muszą zapewniać realizację wspólnego celu w kontekście procesu biznesowego.

Przykłady w opublikowanych artykułach pokazują możliwość stosowania modelu do projektowania rozwiązań integracyjnych wykorzystujących centralny element komunikacji (architektura usługowa, ang. Service-Oriented Architecture) oraz sterowanych w sposób zdecentralizowany (technologia łańcucha bloków, ang. blockchain). Widok *Kontraktów* okazał się predestynowany do opisu inteligentnych kontraktów.

W jednym z ostatnio opublikowanych artykułów habilitant pokazał zastosowanie modelu widoków architektonicznych 1+5 do opisu architektury systemu sterowania bezzałogowym statkiem powietrznym [32]. W artykule wykorzystano dwa widoki architektoniczne: *Przypadków użycia* oraz *Logiczny*. W widoku *Przypadków użycia* umieszczono wymagania funkcjonalne w postaci diagramów przypadków użycia języka UML. W widoku *Logicznym* przedstawiono dopuszczalne stany systemu bezzałogowego statku powietrznego z zastosowaniem diagramów stanów UML. Ciekawym wnioskiem płynącym z pracy jest potrzeba zamodelowania wymagań poza funkcjonalnych, obecnie wyrażonych w formie procedur, np. „REQ 2001: Procedura przekazania". W toku dalszych prac zostanie zamodelowany sposób uwzględniania tego typu wymagań w widoku *Kontraktów* oraz ich bezpośrednie powiązanie z przypadkami testowymi. Model 1+5 zawiera również autorski widok architektoniczny *Integrowanych usług*. Posłuży on do modelowania współpracy bezzałogowego statku powietrznego z naziemną stacją kontroli.

Model 1+5 wspiera także konstrukcję oprogramowania aplikacji rozproszonych sieci łańcucha bloków w podejściu ciągłego dostarczania (ang. Continuous Delivery) oraz ciągłego wdrażania (ang. Continuous Deployment). W sieci łańcucha bloków węzły komunikują się i przechowują dane w sposób rozproszony. Każdy węzeł wykonuje tę samą aplikację biznesową, ale działa w odrębnym środowisku wykonawczym. Rozwiązanie tego typu musi zapewniać pakiety wdrożeniowe w aktualnej wersji aplikacji biznesowej z aktualną wersją plików konfiguracyjnych wdrożenia dla każdego węzła sieci łańcucha bloków [33], [34]. Wykorzystywane są wtedy następujące widoki architektoniczne modelu 1+5: *Kontraktów*, *Logicznym* oraz *Wdrożeniowym*. Wkładem habilitanta jest także propozycja skrótów anglojęzycznych dla trzech pokrewnych praktyk ciągłych, systematyzująca ich nazewnictwo. Dla ciągłej integracji (ang. Continuous Integration) wprowadzono skrót CI. Dla ciągłego dostarczania (ang. Continuous Delivery) wprowadzono skrót CD. Natomiast, dla ciągłego wdrażania (ang. Continuous Deployment) wprowadzono skrót CDT.

Ponadto, cytowania wskazują na przydatność modelu 1+5 w projektowaniu rozwiązań m.in. w obszarze energii odnawialnej, zastosowaniach medycznych oraz transportowych. Generalnie, temat opisu architektury systemów informatycznych rozwijany jest od wielu lat. Jednak aspekt komunikacji pomiędzy systemami nie jest zbyt często uwzględniany. Ostatnio Kirpitsas i in. [35] podkreślili znaczenie modelu widoków architektonicznych 4+1 [18]. Autorzy ci wskazali jednak model widoków architektonicznych 1+5 jako przykład znaczącego postępu, jaki dokonał się w opisie architektury oprogramowania. Podobnie Suljkanovič i in. [36] podkreślili, że to model 1+5 zawiera widoki, które pozwalają na różnorodny opis

architektury systemu informatycznego. Model składa się z dwóch widoków kluczowych dla modelowania przepływów komunikatów: *Integrowanych procesów* oraz *Integrowanych usług*. Z punktu widzenia komunikacji, Zhang i Kianfar [37] przy projektowaniu systemu komunikacji pojazd-infrastruktura wykorzystali widoki *Fizyczny* i *Komunikacja* z podręcznika „Architecture Reference for Cooperative and Intelligent Transportation" [38]. Autorzy ci odwołują się jednak do modelu 1+5, omawiając znaczenie projektowania interoperacyjności pomiędzy różnymi podsystemami. Jeśli chodzi o opiekę zdrowotną, Marbouh i in. [39] budując ramy regulacyjne dla m-zdrowia, położyli nacisk na zaprojektowanie wymiany informacji pomiędzy zainteresowanymi stronami również odwołując się do modelu 1+5. Należy zauważyć, że model 1+5 został także zarekomendowany do opisu architektury oprogramowania i integracji systemu w pracy, Ahmed i in. [40]. Autorzy ci zaproponowali motywacyjny model zaufania oparty na technologii łańcucha bloków dla samochodowych sieci ad hoc (ang. Vehicular Ad hoc NETwork, VANET). Ponadto, model 1+5 został opisany w pracy poświęconej kluczowym aspektom i wyzwaniom w projektowaniu systemów dla środowisk prosumentów energii odnawialnej [41].

IV.5. Literatura

[1] AdapT v2.0 Smart Contract Design Pattern, GitHub Repository. Available online: https://github.com/drGorski/AdapT/releases/tag/v2.0

[2] Daraghmi, E.; Zhang, C.-P.; Yuan, S.-M. Enhancing Saga Pattern for Distributed Transactions within a Microservices Architecture. Applied Sciences, **2022**, 12, 6242. https://doi.org/10.3390/app12126242

[3] Livaja, I.; Pripužić, K.; Sovilj, S.; Vuković, M. A distributed geospatial publish/subscribe system on Apache Spark. Future Generation Computer Systems, **2022**, 132, 282–298. https://doi.org/10.1016/j.future.2022.02.013

[4] Martinez, H.F.; Mondragon, O.H.; Rubio, H.A.; Marquez, J. Computational and Communication Infrastructure Challenges for Resilient Cloud Services. Computers, **2022**, 11, 118. https://doi.org/10.3390/computers11080118

[5] Zhong, Y.; Zhu, S.; Wang, Y.; Li, J.; Zhang, X.; Shang, J.S. Pairwise Location-Aware Publish/Subscribe for Geo-Textual Data Streams. IEEE Access, **2020**, 8, 211704--211713. https://doi.org/10.1109/ACCESS.2020.3038921

[6] Hohpe, G.; Woolf, B. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions; Addison-Wesley Professional: Boston, MA, USA, **2004**.

[7] Messaging Patterns of Enterprise Integration Patterns. Available online: https://www.enterpriseintegrationpatterns.com/patterns/messaging/toc.html

[8] GitHub repository the UML Profile for Messaging Patterns. Available online: https://github.com/drGorski/UMLProfile4MessagingPatterns

[9] GitHub repository of the Symmetric Test Pattern. Available online: https://github.com/drGorski/SymmetricTestPattern

[10] GitHub repository of the UML Profile for Smart Contracts. Available online: https://github.com/drGorski/UMLProfile4SmartContracts

[11] GitHub repository of the UML Profile for Distributed Ledger Deployment. Available online: https://github.com/drGorski/UMLProfileForDLT

[12] GitHub repository of the UML2Deployment Transformation. Available online: https://github.com/drGorski/UML2Deployment

[13] GitHub repository of the UML2Deployment Plugin Transformation. Available online: https://github.com/drGorski/UML2DeploymentPlugin

[14] GitHub repository with the transformation validation project, Available online: https://github.com/drGorski/UML2DeploymentCheck

[15] GitHub repository with the UML Profile for Integration Flows. Available online: https://github.com/drGorski/UMLProfileForIntegrationFlows/blob/master/EIP.epx

[16] GitHub Repository with the Smart Contract Design Pattern Implementation. Available online: https://github.com/drGorski/renewableEnergyBlockchain

[17] ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000). ISO/IEC/IEEE Systems and Software Engineering—Architecture Description. 2011, pp. 1–46. Available online: https://doi.org/10.1109/IEEESTD.2011.6129467

[18] Kruchten, P. Architectural Blueprints—The 4+1 View Model of Software Architecture. IEEE Softw. 1995, 12, 42–50. https://doi.org/10.1109/52.469759

[19] Górski, T. Architectural view model for an integration platform, KKIO 2012 Software Engineering Conference, AGH University of Science and Technology, Cracow, Poland, **2012**, http://kkio2012.agh.edu.pl/presentations/KKIO2012-W1_3.pdf

[20] Górski, T., Architectural view model for an integration platform, Journal of Theoretical and Applied Computer Science, **2012**, 6(1), 25– 34, citeseerx.ist.psu.edu

[21] Górski, T., Platformy integracyjne. Zagadnienia wybrane, Wydawnictwo Naukowe PWN, Warszawa, 2012; ISBN 978-83-01-17071-4.

[22] Górski, T. Verification of Architectural Views Model 1+5 Applicability. In: Moreno-Díaz, R., Pichler, F., Quesada-Arencibia, A. (eds) Computer Aided Systems Theory –

EUROCAST 2019. Lecture Notes in Computer Science, vol. 12013. Springer, Cham, **2020**. https://doi.org/10.1007/978-3-030-45093-9_60,

[23] Górski, T.; Bednarski, J. Modeling of Smart Contracts in Blockchain Solution for Renewable Energy Grid. In: Moreno-Díaz, R., Pichler, F., Quesada-Arencibia, A. (eds) Computer Aided Systems Theory – EUROCAST 2019. Lecture Notes in Computer Science, vol. 12013. Springer, Cham, **2020**. https://doi.org/10.1007/978-3-030-45093-9_61

[24] ZeroMQ Guide, Publish-Subscribe Messaging Patterns, Available online: https://zguide.zeromq.org/docs/chapter5/

[25] ZeroMQ Guide, Request-Reply Messaging Patterns, Available online: https://zguide.zeromq.org/docs/chapter4/

[26] Apache Camel - Enterprise Integration Patterns, Available online: https://camel.apache.org/components/3.20.x/eips/enterprise-integration-patterns.html

[27] Khan, F.; David, I.; Varro, D.; McIntosh, S. Code Cloning in Smart Contracts on the Ethereum Platform: An Extended Replication Study. IEEE Trans. Softw. Eng., **2022**, 49, 2006-2019. https://doi.org/10.1109/TSE.2022.3207428

[28] Solana Network. Available online: https://solana.com

[29] The SmarTS Java package - GitHub repository. Available online: https://github.com/drGorski/SmarTS/releases/tag/v1.1

[30] Elsevier GitHub repository for The SmarTS Java package. Available online: https://github.com/ElsevierSoftwareX/SOFTX-D-24-00045

[31] J. Spray, R. Sinha, A. Sen, X. Cheng, Building Maintainable Software Using Abstraction Layering, IEEE Transactions on Software Engineering 48(11) (2022) 4397-4410. https://doi.org/10.1109/TSE.2021.3119012

[32] Górski, T.; Stecz, W. A Method for Modeling and Testing Near-Real-Time System Scenarios. Applied Sciences, 2024, 14(5), 2023. https://doi.org/10.3390/app14052023

[33] Górski, T., Continuous Delivery of Blockchain Distributed Applications, Sensors 2022, 22(1) 128. https://doi.org/10.3390/s22010128

[34] Górski, T., Towards Continuous Deployment for Blockchain, Applied Sciences, 2021, 11(24), 11745. https://doi.org/10.3390/app112411745

[35] I. K. Kirpitsas and T. P. Pachidis, Evolution towards Hybrid Software Development Methods and Information Systems Audit Challenges, Software, vol. 1, no. 3, pp. 316-363, **2022**, https://doi.org/10.3390/software1030015

[36] A. Suljkanovič, B. Milosavljevič, V. Inđič, and I. Dejanovič, Developing Microservice-Based Applications Using the Silvera Domain-Specific Language, Appl. Sci., vol. 12, no. 13, p. 6679, **2022**, https://doi.org/10.3390/app12136679

[37] K. Zhang and J. Kianfar, An Automatic Incident Detection Method for a Vehicle-to-Infrastructure Communication Environment: Case Study of Interstate 64 in Missouri, Sensors, vol. 22, no. 23, p. 9197, **2022**, https://doi.org/10.3390/s22239197

[38] United States Department of Transportation, March 2023, ``Architecture Reference for Cooperative and Intelligent Transportation," ARC-IT version 9.1, Available online: https://www.arc-it.net/

[39] D. Marbouh, M. C. E. Simsekler, K. Salah, R. Jayaraman, S. Ellahham, A Blockchain-Based Regulatory Framework for mHealth, Data, vol. 7, no. 12, p. 177, **2022**, https://doi.org/10.3390/data7120177

[40] W. Ahmed, W. Di, and D. Mukathe, A Blockchain-Enabled Incentive Trust Management with Threshold Ring Signature Scheme for Traffic Event Validation in VANETs, Sensors, vol. 22, no. 17, p. 6715, **2022**, https://doi.org/10.3390/s22176715

[41] Yiasoumas, G.; Berbakov, L.; Janev, V.; Asmundo, A.; Olabarrieta, E.; Vinci, A.; Baglietto, G.; Georghiou, G.E. Key Aspects and Challenges in the Implementation of Energy Communities. Energies **2023**, 16, 4703. https://doi.org/10.3390/en16124703

V. Informacja o wykazywaniu się istotną aktywnością naukową albo artystyczną realizowaną w więcej niż jednej uczelni, instytucji naukowej lub instytucji kultury, w szczególności zagranicznej.

Habilitant pracował w trzech jednostkach naukowych i w każdej z nich publikował artykuły w czasopismach naukowych.

W trakcie pracy na Wydziale Cybernetyki, Wojskowej Akademii Technicznej (WAT) habilitant publikował głównie w trzech czasopismach: *Biuletyn Wojskowej Akademii Technicznej* (ISSN 1234-5865), *Journal of Theoretical and Applied Computer Science* (ISSN 2299-2634, e-ISSN 2300-5653) oraz *Roczniki Kolegium Analiz Ekonomicznych* (ISSN 1232-4671). Poniżej wykaz wybranych opublikowanych artykułów:

- Górski, T. The use of Enterprise Service Bus to transfer large volumes of data. Journal of Theoretical and Applied Computer Science, **2014**, 8(4), 72–81.
- Górski Tomasz, Unifikacja procesów biznesowych w sektorze medycznym, Roczniki Kolegium Analiz Ekonomicznych, **2014**, no. 35, pp.106-129.

- Górski, T. Model-to-model transformations of architecture descriptions of an integration platform. Journal of Theoretical and Applied Computer Science, **2014**, 8(2), 48–62.
- Górski, T. Model-Driven Development in implementing integration flows. Journal of Theoretical and Applied Computer Science, **2013**, 9(2), 66–82.
- Górski Tomasz, Symulacyjne środowisko badania wydajności platformy integracyjnej rejestrów medycznych Roczniki Kolegium Analiz Ekonomicznych, **2013**, no. 29, pp.595-610.
- Górski, T. UML profiles for architecture description of an integration platform. Bulletin of the Military University of Technology, **2013**, LXII(2), 43–56.
- Górski, T. Architectural view model for an integration platform. Journal of Theoretical and Applied Computer Science, **2012**, 10, 25–34.

Wszystkie wyżej wymienione artykuły zarejestrowane są w Bazie Wiedzy Uniwersytetu Gdańskiego: https://repozytorium.bg.ug.edu.pl

Habilitant opublikował także książkę wydaną przez Państwowe Wydawnictwo Naukowe PWN:

- Górski, T., Platformy integracyjne. Zagadnienia wybrane. Wydawnictwo Naukowe PWN, ISBN: 978-83-01-17071-4, **2012**.

Wszystkie wymienione pozycje, opublikowane w ramach pracy w WAT, dotyczą tematyki rozprawy habilitacyjnej.

W trakcie pracy w Akademii Marynarki Wojennej (AMW) habilitant uczestniczył w konferencjach międzynarodowych, gdzie prezentował wyniki swoich prac badawczych:

- Górski, T. Verification of Architectural Views Model 1+5 Applicability. In: Moreno-Díaz, R., Pichler, F., Quesada-Arencibia, A. (eds) Computer Aided Systems Theory – EUROCAST 2019. Lecture Notes in Computer Science, vol. 12013. Springer, Cham, **2020**. https://doi.org/10.1007/978-3-030-45093-9_60,
- Górski, T.; Bednarski, J. Modeling of Smart Contracts in Blockchain Solution for Renewable Energy Grid. In: Moreno-Díaz, R., Pichler, F., Quesada-Arencibia, A. (eds) Computer Aided Systems Theory – EUROCAST 2019. Lecture Notes in Computer Science, vol. 12013. Springer, Cham, **2020**. https://doi.org/10.1007/978-3-030-45093-9_61,
- Górski, T.; Bednarski, J. Transformation of the UML Deployment Model into a Distributed Ledger Network Configuration, 2020 IEEE 15th International Conference of System of Systems Engineering (SoSE), Budapest, Hungary, **2020**, pp. 255-260. https://doi.org/10.1109/SoSE50414.2020.9130492,

- Górski, T. Towards Enterprise Architecture for Capital Group in Energy Sector, 2018 IEEE 22nd International Conference on Intelligent Engineering Systems (INES), Las Palmas de Gran Canaria, Spain, **2018**, pp. 000239-000244. https://doi.org/10.1109/INES.2018.8523941,
- Górski, T.; Wojtach, E. Use Case API - design pattern for shared data, 2018 26th International Conference on Systems Engineering (ICSEng), Sydney, NSW, Australia, **2018**, pp. 1-8. https://doi.org/10.1109/ICSENG.2018.8638199,
- Górski, T.; Bednarski, J.; Chaczko, Z. Blockchain-based renewable energy exchange management system, 2018 26th International Conference on Systems Engineering (ICSEng), Sydney, NSW, Australia, **2018**, pp. 1-6. https://doi.org/10.1109/ICSENG.2018.8638165, (praca zrealizowana we współpracy z University of Technology Sydney (UTS) w Australii.)

Habilitant publikował artykuły w następujących czasopismach naukowych z listy JCR: *IEEE Access* (e-ISSN 2169-3536), *Symmetry* (ISSN 2073-8994), *Sensors* (ISSN 1424-8220), *Applied Sciences* (ISSN 2076-3417) oraz *International Journal of Electronics and Telecommunications* (eISSN 2300-1933).

Następujące opublikowane w trakcie pracy w AMW artykuły weszły w skład jednotematycznego cyklu publikacji: [A5], [A6], [A7], [A8], [A9].

Artykuły [A7] oraz [A8] zostały opracowane w ramach, kierowanej przez habilitanta, pracy statutowej „*Model widoków architektonicznych współpracujących systemów informatycznych*". Propozycję tej pracy statutowej habilitant zgłosił, uzyskał zgodę Rektora AMW oraz przyznano mu środki na jej przeprowadzenie.

Praca „Optimization of business process execution in services architecture: a systematic literature review" ([A8]) została zrealizowana we współpracy z WAT.

W trakcie pracy w AMW, habilitant opublikował jeszcze trzy artykuły naukowe:

- Górski, T., Continuous Delivery of Blockchain Distributed Applications, Sensors **2022**, 22(1) 128. https://doi.org/10.3390/s22010128
- Górski, T., Towards Continuous Deployment for Blockchain, Applied Sciences, **2021**, 11(24), 11745. https://doi.org/10.3390/app112411745

– Górski, T.; Bednarski, J. Modeling of Distributed Ledger Deployment View, International Journal of Electronics and Telecommunications, **2020**, 66(4), pp. 619–625. https://doi.org/10.24425/ijet.2020.134020

Artykuły te stanowią dodatkowe osiągnięcie. Prace te koncentrują się na widoku *Wdrożenia* sieci łańcucha bloków oraz widoku *Logicznym*. Habilitant zaproponował metodę i zaprojektował mechanizm ciągłego wdrażania (ang. Continuous Deployment) aplikacji rozproszonych technologii łańcucha bloków obejmujący: kod źródłowy inteligentnych kontraktów oraz konfiguracje wdrożeniowe węzłów sieci łańcucha bloków. W rozwiązaniu tym istotne jest ujęcie obydwu aspektów charakterystycznych dla rozwiązań sieci łańcucha bloków. Pierwszym jest aplikacja rozproszona technologii łańcucha bloków. Drugim są konfiguracje wdrożeniowe dla poszczególnych węzłów sieci łańcucha bloków. Zakładano wykorzystanie środowiska kontenerów Kubernetes do utrzymywania sieci łańcucha bloków. Zapewniałoby to potencjalne łatwe wdrażanie nowej konfiguracji węzłów jak i aplikacji rozproszonej w nowej wersji. Jednakże utracona byłaby niezależność węzłów sieci łańcucha bloków, ponieważ węzły mogłyby zostać osadzone na tej samej fizycznej maszynie.

W trakcie pracy w Uniwersytecie Gdańskim (UG) habilitant publikował w następujących czasopismach: *SoftwareX* (e-ISSN 2352-7110), *IEEE Access* (e-ISSN 2169-3536) oraz *Applied Sciences* (ISSN 2076-3417).

Następujące opublikowane w trakcie pracy w UG artykuły weszły w skład jednotematycznego cyklu publikacji: [A1], [A2], [A3], [A4].

Ponadto, habilitant opublikował pracę „A Method for Modeling and Testing Near-Real-Time System Scenarios" ([32]), która została zrealizowana we współpracy z Uniwersytetem im. Kardynała Stefana Wyszyńskiego.

Wszystkie wyżej wymienione artykuły, opublikowane w ramach pracy w UG, dotyczą tematyki rozprawy habilitacyjnej.

VI. Informacja o osiągnięciach dydaktycznych, organizacyjnych oraz popularyzujących naukę lub sztukę.

W trakcie pracy w uczelniach, habilitant w każdej z nich pełnił funkcje organizacyjne. Dodatkowo w WAT prowadził projekt inwestycyjny finansowany z Programu Operacyjnego Infrastruktura i Środowisko (POIŚ). W wyniku pomyślnej realizacji tego projektu powstało *Centrum Studiów Zaawansowanych Inżynierii Systemów*.

Podsumowanie pełnionych funkcji organizacyjnych przedstawiono w Tabela 18.

Tabela 18. Funkcje organizacyjne.

Okres	Funkcja	Jednostka naukowa
01.10.2023– obecnie	Kierownik Zakładu Inżynierii Oprogramowania	Instytut Informatyki, Wydział Matematyki, Fizyki i Informatyki, Uniwersytet Gdański
23.10.2019–22.02.2022	Kierownik Zakładu Systemów Informatycznych	Katedra Informatyki, Wydział Mechaniczny, Akademia Marynarki Wojennej
01.09.2016–30.11.2016	Prodziekan ds. rozwoju i współpracy Wydziału Cybernetyki	Wydział Cybernetyki, Wojskowa Akademia Techniczna
01.01.2012–31.12.2015	Kierownik projektu POIS.13.01.00-00-007/12 pt. Przebudowa budynku nr 65 na cele Centrum Studiów Zaawansowanych Inżynierii Systemów WAT w Warszawie.	Wydział Cybernetyki, Wojskowa Akademia Techniczna
01.09.2012–31.08.2016	Kierownik Zakładu Inżynierii Systemów Informatycznych (w chwili objęcia – Zakładu Metodyk i Narzędzi Informatyki)	Instytut Systemów Informatycznych, Wydział Cybernetyki, Wojskowa Akademia Techniczna
kadencja 2012-2016	Członek Senatu Wojskowej Akademii Technicznej	Wojskowa Akademia Techniczna
kadencja 2012-2016	Członek Komisji ds. Ekonomicznych Rady Wydziału Cybernetyki	Wydział Cybernetyki, Wojskowa Akademia Techniczna

Artykuły opublikowane w wyniku realizacji projektu POIS.13.01.00-00-007/12:

- Górski T., Stryga M., Center for Advanced Studies in Systems Engineering. In: Klempous R., Nikodem J. (eds) Smart Innovations in Engineering and Technology. ICACON 2017, APCASE 2017. Topics in Intelligent Engineering and Informatics, vol 15. Springer, Cham, **2019**, https://doi.org/10.1007/978-3-030-32861-0_1
- Górski, T. Specyfikacja wymagań dla chmury akademickiej w Centrum Studiów Zaawansowanych Inżynierii Systemów WAT, Roczniki Kolegium Analiz Ekonomicznych Szkoły Głównej Handlowej, Zeszyt 46, **2017**, pp. 65-76. https://rocznikikae.sgh.waw.pl/p/roczniki_kae_z46_05.pdf

- Górski, T. Zarządzanie projektem budowy Centrum Studiów Zaawansowanych Inżynierii Systemów WAT, Roczniki Kolegium Analiz Ekonomicznych Szkoły Głównej Handlowej, Zeszyt 46, **2017**, pp. 77-90. https://rocznikikae.sgh.waw.pl/p/roczniki_kae_z46_06.pdf

Habilitant zorganizował i pełnił funkcję Przewodniczącego Komitetu Organizacyjnego Konferencji Naukowej pt. "Systems Engineering 2012", która odbyła się w Warszawie w dniu 19 września 2012 r.

W trakcie pracy w WAT habilitant wypromował kilkudziesięciu magistrów oraz inżynierów informatyki. Poniżej przedstawiono tematy wybranych prac dyplomowych, których promotorem był habilitant w okresie 2007-2012:

- Adrian P. Woźniak, Projekt środowiska automatyzacji modelowania architektury platformy integracyjnej, 2012,
- Jakub Bednarski, Projekt aplikacji z wymianą dokumentów przez platformę integracyjną, 2012,
- Michał Bastrzyk, Metoda testowania platformy integracyjnej, 2012,
- Karol Dowbecki, Metoda zarządzania usługami platformy integracyjnej, 2012,
- Damian Kowalewski, Projekt środowiska do wsparcia procesu projektowania platformy integracyjnej, 2012,
- Kamil Misiak, Metoda włączania systemu informatycznego do platformy integracyjnej, 2012,
- Michał Warecki, Metoda projektowania platformy integracyjnej, 2012,
- Katarzyna Balkiewicz, Konfiguracja procesu projektowania platformy integracyjnej, 2012,
- Katarzyna Ciereszko, Zastosowanie wzorców Java EE w budowaniu wydajnych aplikacji, 2012,
- Iwona Dziuk, Wdrożenie modelu procesu biznesowego na platformie integracyjnej, 2012,
- Krzysztof Kaczmarczyk, Projekt architektury systemu informatycznego z dostępem przez różnego typu urządzenia mobilne, 2011,
- Paweł Napiórkowski, Metoda modelowania architektury platformy integracyjnej, 2011,
- Przemysław Popławski, Projekt środowiska do badania wydajności platform integracyjnych, 2011,
- Iwona Białek, Metoda zarządzania zmianą w projekcie informatycznym, 2009,
- Jakub Olszewski, Metoda iteracyjnej budowy systemu informatycznego w architekturze J2EE, 2009,
- Lidia Pilichowska, Metoda zarządzania testami w projekcie informatycznym realizowanym w sektorze publicznym, 2009,
- Wojciech Rudziński, Projekt rozszerzalnego systemu informatycznego w architekturze J2EE, 2009,
- Grzegorz Zabielski, Projekt skalowalnego systemu informatycznego w architekturze J2EE, 2009,
- Piotr Czajkowski, Projekt środowiska do testowania aplikacji internetowych, 2008,
- Daniel Grzegrzółka, Symulacyjna metoda badania efektywności funkcjonowania aplikacji internetowych, 2008,
- Jacek Kamiński, Projekt systemu informatycznego dla banku z bezpiecznym przetwarzaniem transakcji, 2008,
- Krzysztof Kot, Metoda zapewnienia efektywności funkcjonowania portalu internetowego, 2008,

- Marta Lignowska, Metoda efektywnego zarządzania wymaganiami na system informatyczny dla banku, 2008,
- Artur Lipski, Metoda inkrementalnego rozwijania systemu informatycznego, 2008,
- Krzysztof Nakielski, Metoda badania efektywności procesu wytwórczego oprogramowania, 2008,
- Łukasz Piątkowski, Metoda harmonogramowania projektu informatycznego, 2008,
- Paweł Rzymkiewicz, Metoda tworzenia systemów informatycznych wspierających procesy biznesowe przedsiębiorstwa, 2008,
- Adrian Sędzikowski, Metoda integracji systemów informatycznych przedsiębiorstwa, 2008,
- Waldemar Pac, Projekt portalu korporacyjnego, 2007,
- Błażej Rychlik, Metoda integracji danych systemów informatycznych, 2007,
- Anna Matusik, Metoda przygotowania wdrożenia systemu informatycznego w przedsiębiorstwie, 2007,
- Arkadiusz Jaźwiński, Metoda tworzenia systemów informatycznych wspierających procesy biznesowe przedsiębiorstwa, 2007.

W trakcie pracy w AMW, Rektor tej uczelni w dniu 1 października 2021 roku wyróżnił habilitanta dyplomem za znaczące przedsięwzięcia organizacyjne.

Ponadto, habilitant zaproponował sesję specjalną i pełnił funkcję *Chair of Special Session SS9: "Software Engineering"* na konferencji: *27th International Conference on Systems Engineering*, która odbyła się w Las Vegas, USA, w roku 2020.

VII. Oprócz kwestii wymienionych w pkt. 1-6, wnioskodawca może podać inne informacje, ważne z jego punktu widzenia, dotyczące jego kariery zawodowej.

Habilitant pełni następujące funkcje w czasopismach naukowych z listy JCR:

- Editorial Board Member:
  - IET Software (eISSN 1751-8814); Impact Factor (IF) = 1,6; 70 pkt., https://www.hindawi.com/journals/ietsfw/editors/
  - Open Computer Science (ISSN 2299-1093); IF = 1,5; 40 pkt., https://www.degruyter.com/journal/key/comp/html#editorial
- Topical Advisory Panel Member:
  - Applied Sciences (ISSN 2076-3417); IF = 2,7; 100 pkt., https://www.mdpi.com/journal/applsci/topical_advisory_panel
  - Symmetry (ISSN 2073-8994); IF = 2,7; 70 pkt., https://www.mdpi.com/journal/symmetry/topical_advisory_panel
- Reviewer Editor, Software – Frontiers in Computer Science (eISSN 2624-9898); IF = 2,6; 20 pkt., https://www.frontiersin.org/journals/computer-science/editors

- Reviewer:
  - Information Sciences (ISSN 0020-0255); IF = 8,1; 200 pkt.
  - SoftwareX (eISSN 2352-7110); IF = 3,4; 200 pkt.
  - Journal of Industrial Information Integration (ISSN 2467-964X); IF = 15,7; 140 pkt.
  - Future Generation Computer Systems (eISSN 1872-7115); IF = 7,5; 140 pkt.
  - Scientific Reports (eISSN 2045-2322); IF = 4,6; 140 pkt.
  - Information and Software Technology (eISSN 1873-6025); IF = 3,9; 140 pkt.
  - IEEE Access (e-ISSN 2169-3536); IF = 3,9; 100 pkt.
  - Software and Systems Modeling (eISSN 1619-1374); IF = 2,0; 70 pkt.

Habilitant pełnił funkcję *Guest Editor* dla wydania specjalnego „Advances in Blockchain and Smart Contracts with Diverse Domains Applications" w czasopiśmie Applied Sciences (https://www.mdpi.com/journal/applsci/special_issues/WHLOH2706H). W ramach tego wydania specjalnego opublikowano 2 artykuły, w których procesie recenzowania habilitant pełnił funkcję edytora akademickiego (ang. Academic Editor).

Habilitant pełnił funkcję edytora dla 11 manuskryptów naukowych zgłoszonych do czasopism *IET Software* oraz *Applied Sciences*. Ponadto, habilitant zrecenzował ponad 80 prac zgłoszonych do czasopism naukowych z listy JCR. Potwierdzone zapisy edycji oraz wybranych recenzji artykułów naukowych dostępne są w profilu Web of Science habilitanta: https://www.webofscience.com/wos/author/record/AAJ-1716-2020.

Ponadto, habilitant pełni następujące funkcje w radach konferencji międzynarodowych:

- Program Committee Member – International Conference on Systems Engineering (ICSEng), od 2018 roku, http://www.icseng.com/committees.php
- Technical Committee Member – International Conference on Software and System Engineering (ICoSSE), od 2023 roku, http://www.icsse.org/com.html

Posiadane certyfikaty zawodowe:

- Oracle Certified Associate, Java SE 8 Programmer,
- OMG-Certified UML Professional Fundamental,
- IBM Certified Solution Designer - Rational Unified Process v7.0,
- IBM Certified Solution Designer - Rational Software Architect,
- IBM Certified Instructor Rational Unified Process v. 2003,
- IBM Certified Specialist Rational Unified Process v.2003,
- IBM Certified Instructor for Rational Requirements Management w/Use Cases,
- IBM Certified Specialist for Rational Requirements Management w/Use Cases,

- IBM Certified Instructor for Rational Object-Oriented Analysis and Design v.2003,
- IBM Certified Specialist for Rational Object-Oriented Analysis and Design v.2003.
- IBM Certified Associate Developer - Rational Application Developer for WebSphere Software V6.0,
- IBM Certified Instructor (SW284) Servlet and JSP Development with IBM Rational Application Developer V6,
- IBM Certified Instructor - SW285 - Developing EJBs with IBM Rational Application Developer V6,
- IBM Certified Instructor - SW244 - Introduction to Java Using IBM Rational Application Developer V6,
- PRINCE2 Foundation,
- ITIL Foundation Certificate in IT Management.

W latach 2007-2020 habilitant prowadził firmę RightSolution™, mającą status Autoryzowanego Centrum Edukacyjnego IBM dla marki Rational. Habilitant prowadził szkolenia i konsultacje w zakresie inżynierii oprogramowania.

.....
(podpis wnioskodawcy)