Semina □ Nr 16
Scientiarum 2017
s. 24–42
DOI: http://dx.doi.org/10.15633/ss.2483

Izabela Bondecka-Krzykowska

Dualna natura programów komputerowych

Pytając o to, czym jest informatyka, spotykamy się z opinią, że jest to dyscyplina nauki zajmująca się tworzeniem programów komputerowych. Systemy komputerowe stały się bowiem nieodłączną częścią naszej codzienności; coraz trudniej wyobrazić sobie rzeczywistość bez różnego typu komputerów i działającego na nich oprogramowania. Powstaje zatem pytanie: czym jest program komputerowy?

Wydaje się, że każdy informatyk wie, co to jest program i w jaki sposób wpływa on na działanie komputera, wiedzę tę bowiem wykorzystuje w praktyce, tworząc i badając oprogramowanie. Jednak niewielu z nich zdaje sobie sprawę, jak ciekawym bytem jest program komputerowy z punktu widzenia filozofa. Rozważmy podstawowe kwestie ontologiczne z nim związane, rozpoczynając od odpowiedzi na to – z pozoru proste – pytanie: czym jest program komputerowy?

1. Definicje programu komputerowego

Z pojęciem programu (i programowania) stykają się już dzieci w szkole podstawowej. Przy przeglądaniu szkolnych podręczników informatyki znajdujemy zazwyczaj określenia programu komputerowego zbliżone do definicji podanej przez Grażynę Kobę:

> Program komputerowy to ułożony w odpowiedniej, logicznie powiązanej kolejności zestaw instrukcji dla procesora mówiących, co ma robić z dostarczonymi informacjami i w jakiej kolejności[1].

Definicja ta traktuje program jako ciąg poleceń (instrukcji) dla komputera (procesora), określających jego zachowanie podczas wykonania programu. Jednak nie każdy taki ciąg skłonni jesteśmy uznać za program komputerowy – tworzące go instrukcje powinny bowiem mieć odpowiednią formę. Jakie zatem cechy powinien posiadać ciąg poleceń, by móc być nazwanym programem?

Naturalne wydaje się określenie programu jako sekwencji symboli opisującej obliczenia zgodnie z pewnymi regułami, zwanymi językiem programowania[2]. Taka jego definicja w naturalny sposób rodzi pytania o to, czym jest ów „język" oraz jaki jest jego związek z samym programem. Najprostszym rozwiązaniem tej kwestii jest stwierdzenie, że język programowania to zbiór reguł (rozumianych syntaktycznie) określających m.in. zasady budowania struktury programu oraz sposób tworzenia wchodzących w jego skład instrukcji. Wydaje się jednak, że jest to zbytnie uproszczenie, sprowadza bowiem program do ciągu napisów, pomijając inne istotne jego aspekty, związane np. z wykonaniem i wykorzystaniem programu.

Można również rozumieć programowanie w sposób czysto pragmatyczny, twierdząc, że jego efekty są narzędziami lub środowiskiem pracy ludzi. Programy tworzy się bowiem w celu rozwiązywania problemów z użyciem komputera. Projektuje się je zatem tak, by mogły jak najlepiej odpowiadać na potrzeby użytkowników[3]. Należy jednak pamiętać, że istnieją obiekty nieuznawane powszechnie za programy, które podpadają pod taką definicję. Zapis algorytmu, np. w postaci schematu blokowego czy pseudokodu, może zawierać instrukcje stworzone dla rozwiązania pewnego problemu

---

[1] G. Koba, *Informatyka. Podstawowe tematy. Podręcznik informatyki dla gimnazjum*, Wrocław–Warszawa 2009, s. 9.

[2] Por. M. Ben-Ari, *Understanding Programming Languages*, Chichester 2006.

[3] Por. C. Floyd, *Outline of a Paradigm Change in Software Engineering*, w: *Computers and Democracy: A Scandinavian Challenge*, ed. G. Bjerknes, P. Ehn, M. Kyng, K. Nygaard, Hants 1987, s. 191–210.

przez komputer, ale przecież nie każdy taki zapis skłonni jesteśmy nazwać programem. Co zatem wyróżnia programy spośród wszystkich ciągów instrukcji?

Poszukując odpowiedzi na to pytanie, możemy odnieść pojęcie programu do sprzętu, na którym jest on wykonywany. James H. Moore podaje następującą definicję:

> Program komputerowy jest zbiorem instrukcji, które komputer może wykonać (lub co najmniej istnieje efektywna procedura przekształcenia ich do postaci, którą może on wykonać), by zrealizować zadanie[4].

W sformułowaniu tym autor kładzie szczególny nacisk na możliwość wykonania ciągu instrukcji przez maszynę, jaką jest komputer, uzależniając tym samym pojęcie programu od pojęcia komputera. Instrukcje uważane są bowiem za program jedynie wtedy, gdy może wykonać je maszyna. Algorytm, rozumiany jako ciąg instrukcji, możemy traktować jako program tylko wtedy, gdy został on przedstawiony w formie umożliwiającej jego wykonanie przez komputer.

Zaletą definicji Moore'a jest to, że pozwala ona na proste odróżnienie dwóch podstawowych pojęć informatyki: algorytmu i programu. Wymaga ona jednak pewnych precyzacji. Wydaje się, że występujące w niej pojęcie „komputer" powinno się ograniczyć do maszyn obecnie istniejących. W przeciwnym wypadku każdy ciąg instrukcji jest programem, ponieważ nie można wykluczyć możliwości powstania w przyszłości urządzenia odpowiedniego do wykonania tego ciągu. Jednak przyjęcie tego dodatkowego założenia powoduje, że pojęcie programu uzależnione jest od istniejących obecnie komputerów – tym samym status obiektu jako programu może ulegać zmianie. Pewne ciągi instrukcji dzisiaj nie są programami (gdyż nie ma komputerów, na których można je wykonać), ale mogą stać się nimi w przyszłości. I na odwrót – dawne programy komputerowe dzisiaj nimi nie są, gdyż nie ma już maszyn, na których działały. Ten zmie-

[4] J. H. Moore, *Three Myths of Computer Science*, „The British Journal for the Philosophy of Science" 29 (1978) no. 3, s. 214 (jeśli nie zaznaczono inaczej, tłum. własne).

niający się w czasie status obiektów jako programów komputerowych przeczy powszechnej intuicji związanej z tym pojęciem.

Przedstawione powyżej uwagi dotyczące definiowania programu komputerowego pokazują, że niełatwo jest określić cechy i status ontologiczny obiektu nazywanego programem komputerowym. Jest on jednak bytem niezmiernie ciekawym z punktu widzenia filozofa.

Tradycyjnie w filozofii, wśród wielu rodzajów bytów, wyróżnia się dwa podstawowe: abstrakcyjne oraz konkretne. Zazwyczaj nie mamy problemów z zaklasyfikowaniem większości obiektów, z którymi mamy do czynienia na co dzień. Urządzenie, jakim jest komputer, podobnie jak samochód i drzewo, to przykłady bytów konkretnych (fizycznych). Z kolei liczby, zbiory i algorytmy są obiektami abstrakcyjnymi. Gdzie w tym podziale znajduje się miejsce dla programów komputerowych? Czy programy są obiektami konkretnymi (związane są bowiem z działaniem komputerów), czy też abstrakcyjnymi (gdyż są zapisem algorytmów)? Rozwiązanie tej kwestii ma swoje znaczące konsekwencje praktyczne, m.in. prawne, związane z ich ochroną. Inaczej regulowane są bowiem kwestie ochrony własności w przypadku obiektów fizycznych, a innym prawom podlegają dobra niematerialne. A może programy komputerowe są obiektami o szczególnym statusie, a regulacje prawne z nimi związane muszą dopiero powstać?

2. Dualna natura programów komputerowych

Rozwiązanie kwestii związanych ze statusem ontologicznym programów komputerowych warto rozpocząć od wyraźnego odróżnienia programu jako ciągu instrukcji zapisanych w języku programowania od jego wykonania, które jest nieodłącznie związane z maszyną fizyczną. Na rozróżnienie to – prawdopodobnie jako pierwszy – zwrócił uwagę James Fetzer i dlatego nazywa się je często „dwuznacznością Fetzera”[5].

[5] Por. J. Fetzer, *Program Verification: The Very Idea*, „Communications of Association for Computing Machinery” 31 (1988) no. 9, s. 271–280.

Owa dwuznaczność programów komputerowych znalazła swoje odzwierciedlenie w ontologii programów komputerowych stworzonej przez Ammona H. Edena i Raymonda Turnera[6]. Autorzy przedstawili taksonomię obiektów informatyki, dzieląc je na trzy kategorie: *Hardware*, *Metaprogramy* oraz *Programy*. Obiekty tej ostatniej kategorii dzielą się na dwie grupy, odpowiadające rozróżnieniu wprowadzonemu przez Fetzera. Są to: (1) skrypty – byty, które zawierają poprawnie sformułowane instrukcje dla pewnej klasy komputerów cyfrowych i (2) boty – obiekty powstające przez uruchomienie skryptu w konkretnych warunkach fizycznych (nazywane w systemach operacyjnych „wątkami" – ang. *threads*), które są, w odróżnieniu od skryptów, bytami czasowymi.

Programy rozumiane jako napisy, nazywane tutaj skryptami, definiowane są w następujący sposób: „*Skrypt* $_{DEF4}$ Kategoria bytów S_L («S_L jest programem») takich, że *L* jest językiem programowania pełnym w sensie Turinga oraz *S* jest dobrze sformułowanym wyrażeniem w *L*"[7]. Określenie to związane jest z pojęciem języka programowania, który jest pełny w sensie Turinga (*Turing-complete*), tzn. musi zawierać wszystkie konstrukcje potrzebne do zasymulowania uniwersalnej maszyny Turinga. Pozwala to na wykluczenie z kategorii „programy" pewnych przypadków skrajnych. Jeśli bowiem przez język programowania rozumiemy zbiór dobrze sformułowanych instrukcji dla komputera, a każdy obiekt fizyczny można opisać jako komputer w pewnym trywialnym sensie, to np. włącznik światła można rozumieć jako komputer, którego „język programowania" składa się z dwóch instrukcji {ON, OFF}. Warunek pełności w sensie Turinga gwarantuje w szczególności, że język programowania obsługuje nietrywialny zbiór instrukcji.

Skrypty dzielą się dalej na dwie podkategorie: kody maszynowe, czyli skrypty zapisane w języku maszynowym, i kody źródłowe – skrypty zapisane w języku programowania wysokiego poziomu. W pewnym sensie kody źródłowe są bardziej abstrakcyjne niż

---

[6] Zob. A. H. Eden, R. Turner, *Problems in the Ontology of Computer Programs*, Essex 2006.

[7] A. H. Eden, R. Turner, *Problems in the Ontology*…, dz. cyt., s. 4.

kody maszynowe, ponieważ skrypty napisane w języku maszynowym składają się z ciągów instrukcji interpretowanych przez komputery wprost, natomiast kody źródłowe wymagają do ich wykonania zastosowania dodatkowych programów – kompilatorów lub interpreterów.

Drugiemu rozumieniu programu komputerowego, jako obiektu czasowego związanego z komputerem, odpowiada w omawianej ontologii pojęcie „bot". Kategoria „boty" zawiera zarówno proste procesy, generowane np. przez naciśnięcie przycisku na klawiaturze lub w mikrofalówce, jak też bardziej skomplikowane programy. Boty to byty czasowe, generowane poprzez uruchomienie skryptów, przy czym dany skrypt może być użyty do generowania wielu botów jednocześnie.

Przedstawiona przez Edena i Turnera taksonomia obiektów dobrze oddaje ważne z filozoficznego punktu widzenia rozróżnienie programów jako procesów (nazywanych tutaj botami) wykonywanych na konkretnych maszynach od programów rozumianych abstrakcyjnie (czyli skryptów). Powstaje jednak pytanie o naturę związków łączących programy napisy z ich wykonaniami.

Instrukcje tworzące program można rozumieć jako zapis przyszłych stanów maszyny, na której zostanie on uruchomiony. Wtedy programy napisy są pewnego rodzaju planami czynności (aktów fizycznych) zachodzących w komputerze, czyli planami programów procesów. Nasuwa się wtedy stwierdzenie, że p r o g r a m r o z u m i a n y j a k o p r o c e s o r a z j e g o z a p i s s ą r ó ż n y m i m a n i f e s t a c j a m i t e g o s a m e g o o b i e k t u. Czy jednak takie utożsamienie jest właściwe? Analizując inne przykłady planów i obiektów będących ich fizycznymi realizacjami, trudno obronić taki pogląd. Wydaje się bowiem całkowicie nieintuicyjne utożsamienie np. planu wykładu uniwersyteckiego (zapisanego na kartce) z samym tym wykładem (wygłoszonym przed słuchaczami). Nie są to manifestacje tego samego obiektu, ale zupełnie inne byty.

Bliższe intuicji wydaje się stwierdzenie, że p r o g r a m y n a p i s y p o w o d u j ą l u b w y w o ł u j ą p r o c e s y z a c h o d z ą c e w k o m p u t e r z e. Zatem obiekty tekstowe, jakimi są kody maszynowe programów, w pewien sposób wywołują procesy fizyczne. Powstaje jednak pytanie o naturę takiego związku przyczynowego.

Kwestią otwartą pozostaje rozstrzygnięcie, czy obiekty symboliczne (napisy) mogą powodować pewne efekty przyczynowe.

3. Program jako konkretna abstrakcja

Można próbować uniknąć powyższych trudności poprzez zdefiniowanie programu komputerowego jako k o n k r e t n e j a b s t r a k c j i. Posiada on bowiem nośnik zapisu (tekst, który jest abstrakcją) oraz nośnik wykonania (konkretną realizację w półprzewodnikach)[8]. Tekst programu nie jest zatem programem, lecz tylko zapisem algorytmu lub formalnym opisem procesu obliczeniowego. Jest zatem abstrakcją. Tym, co powoduje, że algorytm ten jest wykonywany przez maszynę, nie jest sam tekst programu, lecz jego fizyczna reprezentacja (program zapisany na nośniku), a więc konkret.

Traktując program komputerowy jako konkretną abstrakcję, stajemy przed koniecznością wyjaśnienia, w jaki sposób obiekt może być jednocześnie abstrakcyjny i konkretny. Szukając jego rozwiązania, Timothy Colburn[9] porównuje klasyczny problem filozoficzny relacji pomiędzy umysłem a ciałem do kwestii związku pomiędzy programem jako konkretem i programem jako abstrakcją. Analizuje on monizm i dualizm, traktując je jako możliwe rozwiązania powyższej kwestii.

Można – w duchu monizmu – twierdzić, że program jest jednym bytem, a abstrakcja i konkret to tylko jego aspekty. Otwartym pozostaje jednak problem rozumienia pojęcia „aspekt". Nie jest też do końca jasne, czy lepsze byłoby uznanie programów komputerowych za byty abstrakcyjne (idealistycznie), czy też za byty konkretne (materialistycznie). Wydaje się bowiem, że każdy z tych wyborów powoduje znaczne zubożenie potocznego rozumienia samego terminu „program". Inne rozwiązanie sugeruje nam dualizm, zgodnie z którym program komputerowy jest bytem zarówno abstrakcyjnym, jak

[8] Por. T. R. Colburn, *Philosophy and Computer Science*, New York–London 2000.

[9] Por. T. R. Colburn, *Philosophy and Computer Science*, dz. cyt.

i konkretnym, co oczywiście nie rozwiązuje problemu relacji pomiędzy konkretem a abstrakcją.

Colburn proponuje przyjęcie zmodyfikowanej wersji zasady harmonii przedustawnej. Zgodnie z tym twierdzeniem, sformułowanym przez Gottfrieda Wilhelma Leibniza, ciało i umysł tylko pozornie wpływają na siebie nawzajem – a harmonia w ich działaniu pochodzi od Boga. Colburn pisze:

> Dla problemu abstrakcyjny/konkretny możemy zastąpić Boga programistą, który z jednej strony, przez przedstawienie algorytmu w tekście programu, opisuje świat mnożenia macierzy, zmiany wielkości okien lub nawet rejestrów procesora; ale który z drugiej strony, przez akt zapisania, skompilowania, asemblowania i linkowania, powoduje ciąg zmian stanów fizycznych, które pasują strukturalnie do jego abstrakcyjnego świata[10].

Poszukując odpowiedzi na pytanie dotyczące związku pomiędzy zapisem programu a jego wykonaniem, można zwrócić się nie tylko ku filozofii umysłu (analizując związek ciała i umysłu), ale również ku filozofii muzyki.

4. Program jako obiekt quasi-partykularny

Program komputerowy wydaje się mieć wiele wspólnego z utworem muzycznym. Kompozytor, tworząc utwór, zapisuje go bowiem – np. na papierze – w postaci partytury, podobnie jak to czyni programista, przedstawiając program w postaci ciągu instrukcji. Partytura jest pewnego rodzaju zbiorem wytycznych dla wykonujących utwór muzyków, podobnie jak program jest ciągiem instrukcji dla wykonującego go komputera. Co więcej, utwory muzyczne są wykonywane na instrumentach, podczas gdy programy – na komputerach. Warto przy tym zauważyć, że zarówno instrumenty muzyczne (włączając struny głosowe człowieka), jak i komputery są obiektami fizycznymi. Przy czym ten sam utwór może być wykonywany wiele razy

[10] T. R. Colburn, *Philosophy and Computer Science*, dz. cyt., s. 208.

(na różnych instrumentach, przez różnych muzyków), podobnie jak program komputerowy może być uruchamiany wielokrotnie na różnych maszynach. Zdarzają się przy tym utwory nigdy niewykonane, podobnie jak programy nigdy nieuruchomione. Ponadto wykonania – zarówno utworów muzycznych, jak i programów – są bytami czasowymi (trwają w czasie). Zatem próbując opisać dualną naturę programów komputerowych, warto zwrócić się ku filozofii muzyki. Co mówią filozofowie muzyki na temat związków partytury utworu muzycznego z licznymi jego wykonaniami?

Anna Brożek, podkreślając unikatowość i szczególny status ontologiczny obiektów, jakimi są utwory muzyczne, pisze:

> Partytura – to nie utwór, lecz jego zapis; utwór muzyczny jest sensem znaków w partyturze zawartych, podobnie jak powieść czy wiersz są sensem odpowiednich słów.
>
> Wykonanie utworu muzycznego nie jest samym utworem, lecz jego realizacją. (…)
>
> Wszystko wskazuje na to, że utworów muzycznych nie da się utożsamić z niczym przestrzennym[11].

Odnosząc powyższe stwierdzenia do programów komputerowych, można stwierdzić, że ich kody źródłowe lub kody maszynowe to nie programy, lecz tylko zapisy programów (podobnie jak partytura jest zapisem utworu muzycznego). Wskazuje to na konieczność wyraźnego odróżnienia programów od nośników, na których zapisane są ich kody. Wykonanie programu – które można nazywać (za Edenem i Turnerem) botem – nie jest programem, lecz jego realizacją. Natomiast sam program jest – w powyższym rozumieniu – sensem symboli tworzących instrukcje wchodzące w skład jego kodu. Powstaje zatem pytanie: jakiego rodzaju obiektem jest tak postrzegany program komputerowy?

Niektórzy filozofowie twierdzą, że utwory muzyczne to uniwersalia, a ich partytury wyznaczają pewne struktury, które realizują wszystkie

---

[11] A. Brożek, *Filozofia nowej muzyki – rediviva*, „Semina Scientiarum" (2011) nr 10, s. 13.

adekwatne wykonania tych utworów. Przyjmując, że programy komputerowe są bytami pod wieloma względami podobnymi do utworów muzycznych, można postawić tezę, że programy to uniwersalia, których przykładami są wszystkie poprawne ich wykonania. Wydaje się jednak, że utwory muzyczne, a tym samym programy komputerowe, nie są uniwersaliami w zwykłym sensie – i to przynajmniej z dwóch powodów. Po pierwsze są one, w odróżnieniu od uniwersaliów, bytami czasowymi. Po drugie niektóre teorie dotyczące uniwersaliów zakładają, że powstają one w wyniku abstrakcji z konkretnych przedmiotów – programy natomiast często nie są efektem takiej abstrakcji. Czym zatem są utwory muzyczne? Anna Brożek pisze:

> Są to, powtórzmy, przedmioty nieprzestrzenne, ale zarazem czasowe (kiedyś powstałe), i nieogólne (a więc nie wyabstrahowane z partykulariów), a będące wytworami czynności komponowania, „uwieczniane" w partyturach i realizowane w wykonaniach.
>
> Ze względu na wymienione własności utwory muzyczne (…) zasługują na miano „przedmiotów quasi-partykularnych"[12].

Programy komputerowe, podobnie jak utwory muzyczne, są bytami o wymiarze czasowym, które nie powstają w procesie abstrakcji z partykulariów, są zapisywane w postaci kodu źródłowego i realizowane przez wykonania na komputerach. Można je więc traktować jako kolejny, po utworach muzycznych, przykład przedmiotów „quasi-partykularnych". Jednak takie rozumienie natury programów komputerowych nie jest powszechne. Dużo więcej zwolenników znajduje pogląd, że programy są bytami matematycznymi.

5. Program jako obiekt matematyczny

Wśród informatyków, szczególnie tych wywodzących się z matematyki, znajduje się wielu zwolenników traktowania informatyki jako kolejnej gałęzi matematyki (stosowanej), a więc jako nauki formal-

[12] A. Brożek, *Filozofia nowej muzyki*…, dz. cyt., s. 14.

nej. Programy komputerowe uważają oni za obiekty matematyczne, twierdząc, że napisy tworzące program są wyrażeniami matematycznymi, które z niespotykaną dokładnością opisują zachowanie komputerów, na których są wykonywane[13]. Co więcej, uważają oni, że programy rozumiane jako procesy (*p*) wykonywane na komputerze są również obiektami matematycznymi, ponieważ są one w pełni opisywane przez wyrażenia matematyczne (programy rozumiane jako napisy). Eden pisze:

> Programy napisy są wyrażeniami matematycznymi. Wyrażenia matematyczne reprezentują obiekty matematyczne. Program *p* jest w pełni i dokładnie charakteryzowany przez program napis. A zatem program jest obiektem matematycznym[14].

Szukając uzasadnienia dla tego twierdzenia warto przeanalizować zdania, z których zbudowane są programy komputerowe. Powszechnie uważa się bowiem, że zdania opisujące obiekty nauk formalnych (takich jak matematyka) są zdaniami analitycznymi, w odróżnieniu od zdań syntetycznych, specyficznych dla nauk realnych.

Rozpocznijmy od występującej we wszystkich współczesnych programach instrukcji podstawienia (przypisania), zapisywanej np. w postaci *A*:=2+3. Wyrażenia takie można interpretować na dwa sposoby: jako rozkazy lub jako zdania oznajmujące. W pierwszym przypadku jest to rozkaz dla procesora, nakazujący, by zmiennej *A* przypisać sumę liczb 2 i 3. Oczywiście rozkazy nie są zdaniami w sensie logicznym, nie mogą więc być ani prawdziwe, ani fałszywe, a co za tym idzie – nie ma podstaw, by zaklasyfikować je jako analityczne bądź syntetyczne. Pozostaje zatem druga możliwość.

Operację podstawienia można interpretować jako zdanie oznajmujące, które stwierdza, że zmienna oznaczana przez *A* otrzymuje wartość odpowiedniej sumy. Zauważmy, że *A* może być przy tym

---

[13] Por. np. C. A. R. Hoare, *An Axiomatic Basis for Computer Programming*, „Communications of the Association for Computing Machinery" 12 (1969) iss. 10, s. 576–580.

[14] A. H. Eden, *Three Paradigms of Computer Science*, „Minds and Machines" 17 (2007) iss. 2, s. 145.

rozumiane jako oznaczenie albo bytu fizycznego, albo też – abstrakcyjnego. Jeśli *A* potraktujemy jako fizyczne miejsce w pamięci komputera, to operację podstawienia rozumiemy jako stwierdzenie: „fizyczne miejsce w pamięci *A* przyjmuje wartość fizycznego obliczenia 2+3". Jest ono więc swego rodzaju opisem tego, co stanie się w pamięci komputera po wykonaniu operacji podstawienia. Jednak operacje zapisane w językach programowania wysokiego poziomu (w tym operacja podstawienia) nie są wykonywane przez procesor wprost, lecz są one tłumaczone za pomocą specjalnych programów na podstawowe instrukcje maszynowe, które wykonywane są przez procesor. A więc zdanie reprezentujące operację podstawienia stanowi tylko p r z y p u s z c z e n i e co do wyniku wykonania programu i jest zdaniem syntetycznym, ponieważ jego prawdziwość zależy od rzeczywistości. To, czy miejsce w pamięci nazywane zmienną *A* rzeczywiście przyjmie wartość sumy liczb 2 i 3, zależy bowiem od poprawności kompilacji (lub interpretacji) instrukcji podstawienia oraz – w sposób oczywisty – od działania komputera jako maszyny.

Powstaje zatem naturalne pytanie, czy potraktowanie zmiennej *A*, występującej w instrukcji podstawienia *A*:=2+3, jako obiektu abstrakcyjnego (miejsca w pamięci maszyny abstrakcyjnej) zmieni status tej operacji. Sprawdźmy, czy instrukcje programów rozumianych jako opisy stanów maszyny abstrakcyjnej są zdaniami analitycznymi. Rozważmy w tym celu prawdziwość zdania: „Abstrakcyjne miejsce w pamięci *A* przyjmuje wartość abstrakcyjnego obliczenia 2+3". Wyrażenie takie podobne jest do matematycznego założenia co do zawartości zmiennej typu „niech a = 965", czyli do twierdzenia matematycznego rozpoczynającego się od „niech", ale bez późniejszego „wtedy"[15]. Trudno mówić o prawdziwości lub fałszywości takiego wyrażenia – po prostu zostało zrobione założenie. Takie zdanie nie jest więc ani analitycznie, ani syntetyczne. Co więcej, ponieważ każde wyrażenie dowolnego języka programowania można rozumieć abstrakcyjnie, to i program komputerowy można traktować jako ciąg warunków w świecie abstrakcyjnym. Ale tak rozumiany program,

[15] Zob. T. R. Colburn, *Philosophy and Computer Science*, dz. cyt.

podobnie jak ciąg matematycznych „niech", nie jest wyrażeniem matematycznym w żadnym interesującym sensie.

Tak więc podsumowując, wyrażenia języków programowania składające się na program komputerowy są albo zdaniami syntetycznymi, albo też wyrażeniami, których nie można traktować jako zdań analitycznych. Nie sposób więc na podstawie ich analizy orzec w sposób jednoznaczny, czy programy komputerowe są obiektami matematycznymi, czy też nie.

Jeśli jednak wbrew powyższej analizie zgodzimy się, że programy są obiektami matematycznymi, to powstaje pytanie o to, jakiego rodzaju są to obiekty.

Programy można traktować jako funkcje (obiekty czysto matematyczne) ze zbioru stanów początkowych (lub „wejść") w zbiór stanów końcowych (lub „wyjść"). Prowadzi to do prostej analogii pomiędzy programowaniem a dowodzeniem w matematyce, reguły wynikania stosowane w dowodach można bowiem rozumieć jako funkcje działające ze zbioru przesłanek do zbioru wniosków. Analogię tę ilustruje poniższa tabela[16].

	Matematyka	Programowanie
Dziedzina	przesłanki	wejście (*input*)
Funkcja	reguły wynikania	program
Przeciwdziedzina	twierdzenia	wyjście (*output*)

Można również interpretować programy „metamatematycznie", jako odpowiedniki twierdzeń, jak to zaproponowali William L. Scherlis i Dana S. Scott[17], przedstawiając następujący związek pomiędzy matematyką a programowaniem:

Matematyka	Programowanie
problem	specyfikacja
stwierdzenie	program
dowód	weryfikacja programu

[16] Zob. J. Fetzer, *Philosophical Aspects of Program Verification*, „Minds and Machines" 1 (1991) iss. 2, s. 200.

[17] Por. W. L. Sherlis, D. S. Scott, *First Steps towards Inferential Programming*, w: *Information Processing 83*, ed. R. E. A. Mason, New York 1983.

Jak pokazuje powyższa tabela, autorzy porównują proces tworzenia programu do procesu dowodzenia twierdzeń matematycznych. W matematyce postawiony problem może prowadzić do sformułowania jego rozwiązania w postaci stwierdzenia (hipotezy), którego prawdziwość weryfikuje się, przeprowadzając odpowiedni dowód. Z kolei w programowaniu specyfikacja określa wymagania stawiane przed programem, czyli w pewien sposób opisuje problem. Sam program jest zapisem rozwiązania tegoż problemu; analogicznie – stwierdzenie jest zapisem rozwiązania problemu matematycznego. Natomiast weryfikacja programu, rozumiana jako formalne sprawdzenie jego poprawności, odpowiada dowodowi stwierdzenia matematycznego, czyli pokazuje, że program jest zgodny ze specyfikacją[18]. Zgodnie z tym rozumowaniem program jest obiektem matematycznym podobnym do twierdzenia matematycznego. Można jednak rozumieć program jako obiekt abstrakcyjny innego rodzaju.

6. Program jako cyfrowy wzorzec

Peter Suber[19] twierdzi, że program komputerowy to po prostu cyfrowy wzorzec, rozumiany jako tablica komórek (lub miejsc) zawierających jeden z dwóch możliwych symboli.

Uzasadniając swoją koncepcję, autor sformułował szereg silnych założeń ontologicznych, nazywając je zasadami. Wśród nich znalazło się m.in. stwierdzenie, że każdy wzorzec analogowy może zostać odtworzony we wzorcu cyfrowym (za s a d a c y f r o w a) oraz że wzorce cyfrowe i analogowe wyczerpują dziedzinę wzorców (za s a d a w y c z e r p y w a n i a). Prowadzi to do stwierdzenia, że „każdy rodzaj wzorca może posłużyć jako program”[20].

---

[18] Więcej informacji na temat sprawdzania poprawności programów znaleźć można m.in. w: I. Bondecka-Krzykowska, *Z zagadnień ontologicznych informatyki*, Poznań 2016.

[19] Por. P. Suber, *What Is Software?*, „Journal of Speculative Philosophy” 2 (1998) no. 2, s. 89–119.

[20] P. Suber, *What Is Software?*, dz. cyt., s. 3.

Takie rozumienie programu komputerowego jest zbyt ogólne. Po pierwsze nie stwierdza, czy wzorzec musi mieć wymiar materialny, czy ma on być napisany na papierze, nagrany na dysku, czy też wystarczy, by został wymyślony; a może w myśl ujęcia platońskiego: coś jest programem nawet jeszcze przed byciem pomyślanym? Po drugie definicja ta nie podaje żadnego kryterium pozwalającego odróżnić programy od szumu (losowego układu zer i jedynek) i od danych (nie wiadomo, w jaki sposób odróżnić aktywną i pasywną rolę wzorców). Co więcej, jeśli przyjąć zaproponowaną przez Subera zasadę bezszumowości, głoszącą, że żaden wzorzec nie jest szumem dla wszystkich możliwych języków programowania i wszystkich możliwych maszyn – to każdy układ zer i jedynek można rozumieć jako program w myśl pewnej interpretacji (w odpowiednim języku programowania i dla odpowiedniej maszyny). Jeśli zasada ta jest prawdziwa, to żaden wzorzec nie jest szumem ze wszystkich możliwych perspektyw. Dodanie dwóch dodatkowych warunków, które muszą spełniać wzorce będące programami, a mianowicie: ich czytelność dla maszyny oraz wykonywalność, nadal nie rozwiązuje problemu odróżnienia programu od szumu. Przecież wzorzec będący szumem może być czytelny i może zostać wykonany przez maszynę, tyle tylko że wykonany źle!

Wydaje się, że dla odróżnienia wzorców będących poprawnie działającymi programami od szumu konieczne jest odwołanie do założeń lub celu programisty. Programy realizują bowiem cele swoich autorów, natomiast kody losowe lub błędne – nie. Jednak pojęcie celu programisty nie jest łatwe do zdefiniowania, szczególnie jeśli zgodzimy się z istnieniem „programów naturalnych", wpisanych w przyrodę, dla których trudno jest nawet określić ich twórcę.

Co więcej, konieczne jest doprecyzowanie pojęcia „czytelności", by móc uznać za program odpowiedni zapis na kartce papieru, który nie jest przecież czytelny dla maszyny. Można by np. przyjąć zasadę dostrzegalności, mówiącą, że możliwe jest fizyczne urzeczywistnienie każdego wzorca, ponieważ wszystko, co ma reprezentację fizyczną (np. zostało narysowane), może być przeczytane lub odkodowane przez odpowiednio zaprojektowaną maszynę. A zatem każdy wzorzec jest czytelny dla jakiejś ma-

szyny, czyli każdy wzorzec jest programem. Reasumując, dodanie do definicji programu jako wzorca wymogu jego czytelności dla maszyny nie rozwiązuje problemu odróżnienia go od szumu, gdyż każdy wzorzec jest programem – w pewnym kontekście. Suber zauważa, że:

> Konsekwencje tego faktu [że wszystkie wzorce są programami – przyp. I. B.-K.] na pierwszy rzut oka są zaskakujące. Nie tylko losowy ciąg bitów jest programem, ale również cyfr liczby Pi, krzywe tworzące usta Mony Lisy, (…) gwiazdy w Drodze Mlecznej, układ neuronów w mózgu. (…) Wszechświat jest programem[21].

Z definicją podaną przez Subera związane są jeszcze dwa inne ważne problemy ontologiczne: rozróżnienie wzorca jako danych od wzorca jako programu oraz określenie kryteriów równości programów. Rozwiązanie pierwszego z nich autor widzi w stwierdzeniu, że wzorce programy to te, które uruchamiane są wcześniej i które pełnią funkcję kontrolną względem wzorców danych. Pozostawia przy tym bez komentarza np. kwestię obliczeń równoległych, gdy trudno jest mówić o pierwszeństwie wykonania.

Problematyczne jest również rozróżnienie programów na podstawie wzorców. Jeśli bowiem rozumieć wzorzec w terminach czysto syntaktycznych, to różne wzorce (różniące się choćby jednym bitem) są różnymi programami. Jednak kod programu w języku maszynowym powstający przez kompilację kodu napisanego w języku programowania wysokiego poziomu to przecież różne wzorce będące tym samym programem. Wydaje się, że do identyfikacji programów potrzebne jest przejście od składni do semantyki (uwzględniającej wspomniane już pojęcie celu programisty). Suber dodaje w tym celu do listy zasad kolejną (którą nazywa z a s a d ą p i t a g o r e j s k ą), głoszącą, że składnia „budzi" semantykę, co pozwala mu na stwierdzenie, że istotą programu jest sam wzorzec. Autor nie wyjaśnia jednak, jak należy rozumieć owo „budzenie".

---

[21] P. Suber, *What Is Software?*, dz. cyt., s. 10.

W świetle powyższych uwag stwierdzić można, że koncepcja programu jako cyfrowego wzorca boryka się z wieloma trudnościami. Autor podaje liczne zasady, będące silnymi założeniami ontologicznymi, zazwyczaj bez żadnego uzasadnienia. Co więcej, używane pojęcie „wzorca cyfrowego" bez bliższego określenia jego natury wydaje się nie mniej problematyczne niż samo pojęcie programu. Jeśli rozumiemy wzorce jako obiekty matematyczne, to definicja Subera jest przecież tylko kolejną, nie do końca przemyślaną i niedostatecznie uzasadnioną, wersją twierdzenia, że programy są obiektami matematycznymi.

Podsumowanie

Stworzenie ontologii programów komputerowych nie jest sprawą łatwą. Można je bowiem rozumieć na wiele sposobów, m.in. jako:

– konkretne abstrakcje,
– obiekty quasi-partykularne (podobne do utworów muzycznych),
– byty matematyczne (różnego rodzaju),
– cyfrowe wzorce.

Należy przy tym pamiętać o dualnej naturze programów komputerowych, określanej jako dwuznaczność Fetzera. Mianem programu określa się bowiem zarówno ciągi instrukcji zapisanych w języku programowania, jak i procesy wykonywane na komputerach. Nadal nierozwiązaną kwestią pozostaje określenie, czy programy są obiektami fizycznymi, czy też bytami abstrakcyjnymi. Jej rozstrzygnięcie jest istotne nie tylko z punktu widzenia filozofii, ale ma również ważne konsekwencje praktyczne, związane m.in. z ochroną prawną programów. A może programy komputerowe należy traktować jako obiekty o szczególnym statusie ontologicznym, które nie są ani konkretne, ani abstrakcyjne? Czy filozofowie znajdą taką charakterystykę obiektów nazywanych programami komputerowymi, która adekwatnie oddawać będzie ich szczególną, dualną naturę?

Summary

Dual nature of computer programs

The paper is devoted to the discussion on ontological status of the computer programs. The most popular conceptions are presented and critically discussed: programs as concrete abstractions, as quasi-particular objects (similar to musical pieces), as mathematical objects (of different kinds), and finally – program as digital pattern. Advantages and disadvantages of those approaches are pointed out and some possible solutions are proposed.

Keywords: philosophy of computer science, ontology of computer programs, quasi-particular objects, program as mathematical object, program as pattern

Bibliografia

Ben-Ari M., *Understanding Programming Languages*, Chichester 2006.

Bondecka-Krzykowska I., *Z zagadnień ontologicznych informatyki*, Poznań 2016.

Brożek A., *Filozofia nowej muzyki – rediviva*, „Semina Scientiarum" (2011) nr 10, s. 10–20.

Colburn T. R., *Philosophy and Computer Science*, New York–London 2000.

Eden A. H., *Three Paradigms of Computer Science*, „Minds and Machines" 17 (2007) iss. 2, s. 135–167.

Eden A. H., Turner R., *Problems in the Ontology of Computer Programs*, Essex 2006.

Fetzer J., *Philosophical Aspects of Program Verification*, „Minds and Machines" 1 (1991) iss. 2, s. 197−216.

Fetzer, J., *Program Verification: The Very Idea*, „Communications of Association for Computing Machinery" 31 (1988) no. 9, s. 271−280.

Floyd C., *Outline of a Paradigm Change in Software Engineering*, w: *Computers and Democracy: A Scandinavian Challenge*, ed. G. Bjerknes, P. Ehn, M. Kyng, K. Nygaard, Hants 1987, s. 191–210.

Hoare C. A. R., *An Axiomatic Basis for Computer Programming*, „Communications of the Association for Computing Machinery" 12 (1969) iss. 10, s. 576−580.

Koba G., *Informatyka. Podstawowe tematy. Podręcznik informatyki dla gimnazjum*, Wrocław–Warszawa 2009.

Moore J. H., *Three Myths of Computer Science*, „The British Journal for the Philosophy of Science" 29 (1978) no. 3, s. 213–222.

Sherlis W. L., Scott D. S., *First Steps towards Inferential Programming*, w: *Information Processing 83*, ed. R. E. A. Mason, New York 1983, s. 199–212.

Suber P., *What Is Software?*, „Journal of Speculative Philosophy" 2 (1998) no. 2, s. 89–119.