Testowanie gier

Konrad Kasprzyk, Tomasz Woszczyński

Zestaw zadań nr 3

Zadanie 1

Narzędzie **CCCC** (C and C++ Code Counter) służy do analizy kodu źródłowego różnych języków (jednak głównie skupia się na C++) oraz generowania raportów w postaci HTML. Korzystanie z programu jest dość proste, w konsoli w katalogu ze skompilowanym CCCC wystarczy wpisać polecenie `cccc [ścieżka]`, np. `cccc E:\dir\*.cpp`. Wszystkie dodatkowe opcje możemy sprawdzić flagą `--help`.

SonarQube służy do kontroli jakości kodu w celu w celu wykrycia błędów, luk w zabezpieczeniach, złych praktyk programowania, braku pokrycia testami itd. Program SonarQube uruchamia się w przeglądarce. Możliwe jest uruchomienie analizy kodu poprzez rozszerzenie lub wtyczkę do edytora tekstu. Jednak to nie pokazuje klarownego podsumowania całego projektu jak w przeglądarce. Projekt do analizy można wgrać z wielu źródeł: lokalnie, GitHub, Azure itd.

Zadanie 2 (CCCC)

Programem analizowanym przez CCCC jest gra The Adventure in Caves (https://github.com/whiskeyo/The-Adventure-in-Caves). Po uruchomieniu polecenia opisanego w zadaniu 1. uzyskujemy cały katalog plików z rozszerzeniami `.html` oraz `.xml`. Z pliku `cccc.html` możemy przejść do kilku różnych statystyk takich jak podsumowanie projektu, metryki procedur, właściwości dotyczących projektowania obiektowego, metryki struktur. Wszystkie wartości opisywane poniżej są dokładniej opisane w dokumentacji.

Pierwsza tabela przedstawia podsumowanie projektu:

Metric	Tag	Overall	Per Module
Number of modules	NOM	30	
Lines of Code	LOC	1037	34.567
McCabe's Cyclomatic Number	MVG	175	5.833
Lines of Comment	COM	55	1.833
LOC/COM	L_C	*18.855*	
MVG/COM	M_C	3.182	
Information Flow measure (inclusive)	IF4	399	13.300
Information Flow measure (visible)	IF4v	297	9.900
Information Flow measure (concrete)	IF4c	93	3.100
Lines of Code rejected by parser	REJ	27	

Liczba modułów (NOM) i linii kodu (LOC) mówią same za siebie i na ich podstawie można stwierdzić, że sam projekt nie jest bardzo duży. Złożoność cyklomatyczna (MVG) mówi o stopniu skomplikowania programu i jest wyliczana na podstawie liczby punktów decyzyjnych w programie, w przypadku gry The Adventure in Caves jest to prawie 6 na moduł, co oznacza że kod jest dość prosty i stwarza nieznaczne ryzyko (kolejne przedziały to 11-20 kod złożony powodujący ryzyko na średnim poziomie, 21-50 to kod bardzo złożony związany z bardzo dużym ryzykiem i powyżej 50 to kod niestabilny grożący bardzo wysokim poziomem ryzyka. Liczba linii komentarzy (COM) jest kolejną statystyką, która przydaje się do wyznaczenia innych wartości: liczby linii kodu na linię komentarza (L_C) oraz złożoności cyklomatycznej na linię komentarza (M_C). Dzięki wartością COM, L_C oraz M_C niestety nie można jednoznacznie stwierdzić, czy kod jest napisany dobrze i czy jest wystarczająco dobrze udokumentowany. Wartości mówiące o mierze przepływu informacji (IF4*) mówią o tym ile informacji przepływa między modułami.

Kolejna tabela przedstawia metryki procedur:

Module Name	LOC	MVG	COM	L_C	M_C
Action	2	0	0	------	------
Block	21	1	0	******	------
Character	13	0	0	------	------
Clock	0	0	0	------	------
Coin	46	4	1	46.000	------
Collectible	6	0	0	------	------
Collider	67	9	1	67.000	*9.000*
Color	0	0	0	------	------
Drawable	0	0	0	------	------
Enemy	88	17	5	*17.600*	3.400
Game	168	38	15	*11.200*	2.533
HUD	47	8	4	*11.750*	2.000
Heart	46	4	1	46.000	------
Player	181	38	9	*20.111*	4.222
Portal	57	8	1	57.000	*8.000*
RectangleShape	0	0	0	------	------
RenderWindow	0	0	0	------	------
ResourceManager	56	5	0	******	******
Sprite	0	0	0	------	------
Text	0	0	0	------	------
Vector2f	0	0	0	------	------
Vector2u	0	0	0	------	------
Window	105	17	4	*26.250*	4.250
World	89	25	4	*22.250*	*6.250*
anonymous	6	1	8	------	------
bool	0	0	0	------	------
list	0	0	0	------	------
map	0	0	0	------	------
string	0	0	0	------	------
vector	0	0	0	------	------

Są to informacje takie jak w poprzedniej statystyce, jednak teraz są one dokładniejsze: mówią o każdym module z osobna, dzięki czemu można wywnioskować złożoność poszczególnych modułów. Na przykład kod zawarty w module `Window` jest w ogóle nieudokumentowany (nie zawiera komentarzy), a w module `Game` liczba MVG jest bardzo wysoka, czyli kod źródłowy jest bardzo złożony i wiąże się z dużym ryzykiem. Podobnie można przeanalizować pozostałe moduły.

Następna tabela mówi o właściwościach dotyczących projektowania obiektowego:

Module Name	WMC1	WMCv	DIT	NOC	CBO
Action	0	0	0	0	3
Block	4	0	0	0	4
Character	9	2	0	2	4
Clock	0	0	0	0	6
Coin	5	0	1	0	6
Collectible	3	0	0	3	4
Collider	6	0	0	0	2
Color	0	0	0	0	1
Drawable	0	0	0	0	1
Enemy	14	0	1	0	7
Game	7	7	0	0	6
HUD	3	2	0	0	6
Heart	5	0	1	0	6
Player	25	0	1	0	11
Portal	7	0	1	0	6
RectangleShape	0	0	0	0	7
RenderWindow	0	0	0	0	11
ResourceManager	8	3	0	0	2
Sprite	0	0	0	0	6
Text	0	0	0	0	1
Vector2f	0	0	0	0	9
Vector2u	0	0	0	0	1
Window	18	1	0	0	6
World	4	0	0	0	5
anonymous	1	0	0	0	0
bool	0	0	0	0	3
list	0	0	0	0	1
map	0	0	0	0	1
string	0	0	0	0	4
vector	0	0	0	0	2

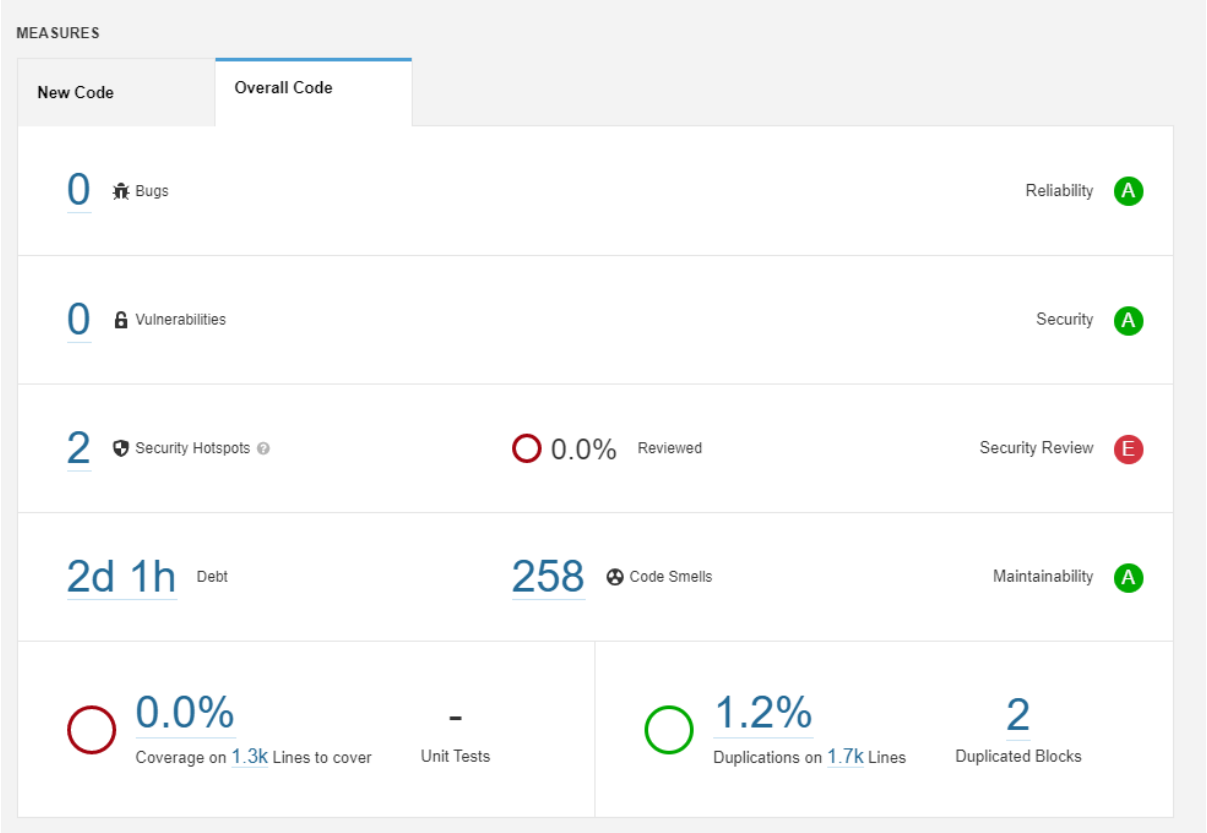
Kolumna WMC1 oznacza liczbę funkcji zawartych w danym module, a kolumna WMCv oznacza liczbę funkcji dostępnych z innych modułów (a więc nie zlicza funkcji prywatnych). Kolumna DIT oznacza głębokość drzewa dziedziczenia - w niektórych przypadkach można zauważyć wartość 1, co oznacza, że moduł dziedziczy z innego (np. klasa `Heart` dziedziczy z klasy `Collectible`), a więc powiązany ze sobą kod jest używany kilkukrotnie. NOC oznacza liczbę dzieci, a więc liczbę modułów bezpośrednio dziedziczących z tego modułu. CBO jest wartością oznaczającą powiązanie między obiektami, mówi ona o wywołaniach funkcji z innych modułów w obrębie wybranego modułu. Wartości w przypadku analizowanej gry są zwykle w okolicach od 5 do 10, a więc moduły współpracują ze sobą w trakcie działania programu.

Następnie mamy podsumowanie metryk strukturalnych:

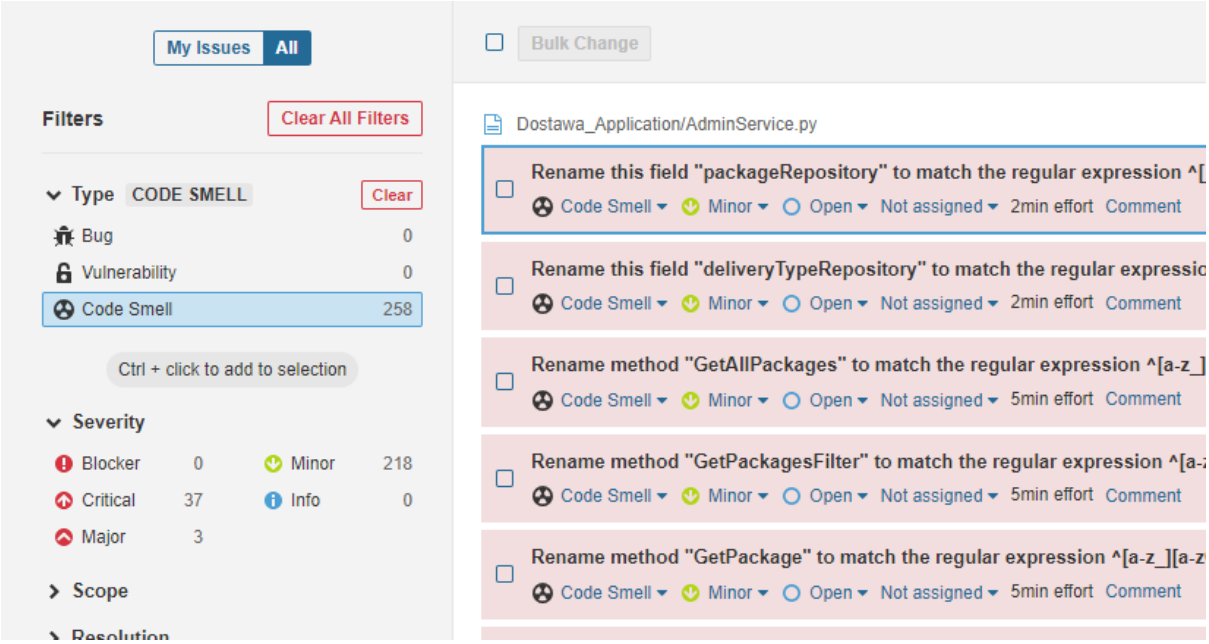
Module Name	Fan-out			Fan-in			IF4		
	vis	con	inc	vis	con	incl	vis	con	inc
Action	2	3	3	0	0	0	0	0	0
Block	0	0	0	4	3	4	0	0	0
Character	2	2	2	2	1	2	16	4	16
Clock	6	6	6	0	0	0	0	0	0
Coin	0	0	0	5	5	6	0	0	0
Collectible	0	3	3	1	0	1	0	0	9
Collider	0	0	0	2	0	2	0	0	0
Color	1	1	1	0	0	0	0	0	0
Drawable	1	0	1	0	0	0	0	0	0
Enemy	0	0	0	*7*	6	7	0	0	0
Game	0	0	0	5	6	6	0	0	0
HUD	1	1	1	5	3	5	25	9	25
Heart	0	0	0	5	5	6	0	0	0
Player	2	1	2	*8*	*8*	9	**256**	*64*	*324*
Portal	0	0	0	5	5	6	0	0	0
RectangleShape	*7*	6	7	0	0	0	0	0	0
RenderWindow	*11*	1	11	0	0	0	0	0	0
ResourceManager	0	0	0	2	2	2	0	0	0
Sprite	6	6	6	0	0	0	0	0	0
Text	1	1	1	0	0	0	0	0	0
Vector2f	*9*	*8*	9	0	0	0	0	0	0
Vector2u	1	1	1	0	0	0	0	0	0
Window	0	1	1	4	4	5	0	16	25
World	0	0	0	5	4	5	0	0	0
anonymous	0	0	0	0	0	0	0	0	0
bool	2	3	3	0	0	0	0	0	0
list	1	1	1	0	0	0	0	0	0
map	1	1	1	0	0	0	0	0	0
string	4	4	4	0	0	0	0	0	0
vector	2	2	2	0	0	0	0	0	0

Liczby Fan-in (FI) oznaczają liczbę modułów przekazujących informacje do bieżącego modułu. Liczby Fan-out (FO) oznaczają liczbę modułów, do których informacje przekazuje bieżący moduł. Miara przepływu informacji (IF4) jest złożoną miarą złożoności strukturalnej i obliczana jest poprzez podniesienie do kwadratu iloczynu FI oraz FO. Wartości vis, con, inc mają następujące znaczenie: licznik części widocznej do innych modułów (a więc funkcji), licznik powiązań z innymi modułami (a więc takich, które wymagają rekompilacji modułu gdy powiązany moduł został zmieniony), licznik wszystkich funkcji. Na podstawie tej tabeli można stwierdzić jak bardzo złożony jest kod i ile informacji przetwarza każdy moduł w stosunku do innych.

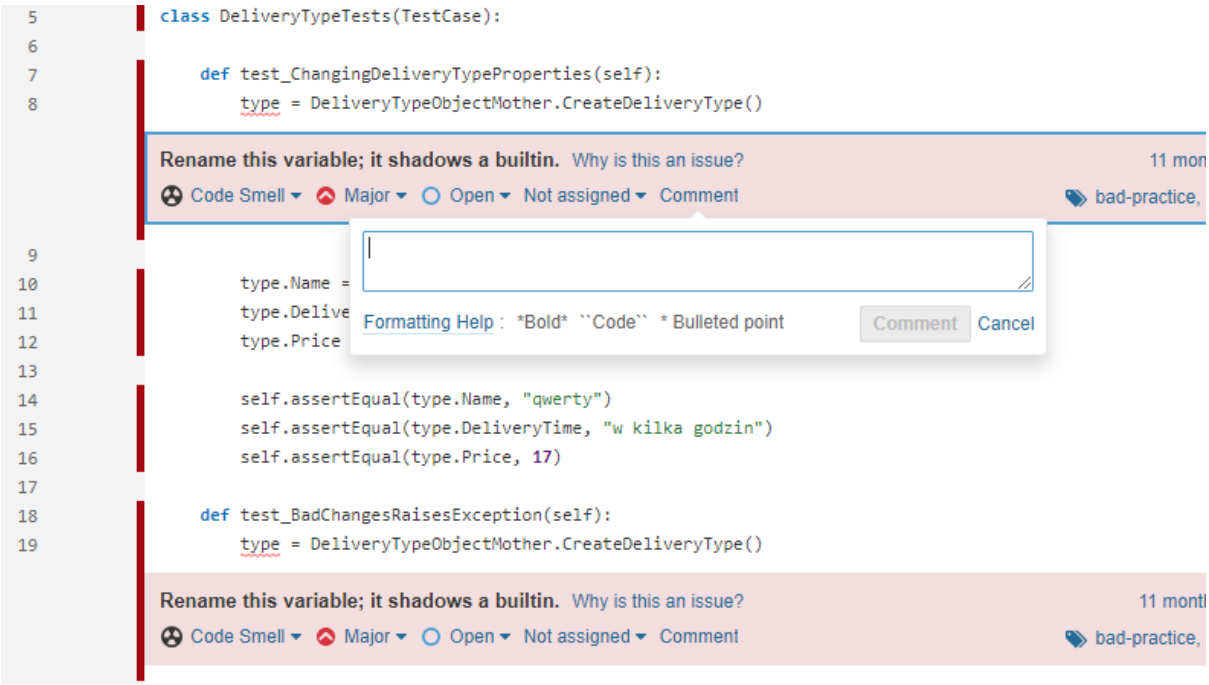
Zadanie 2 (SonarQube)



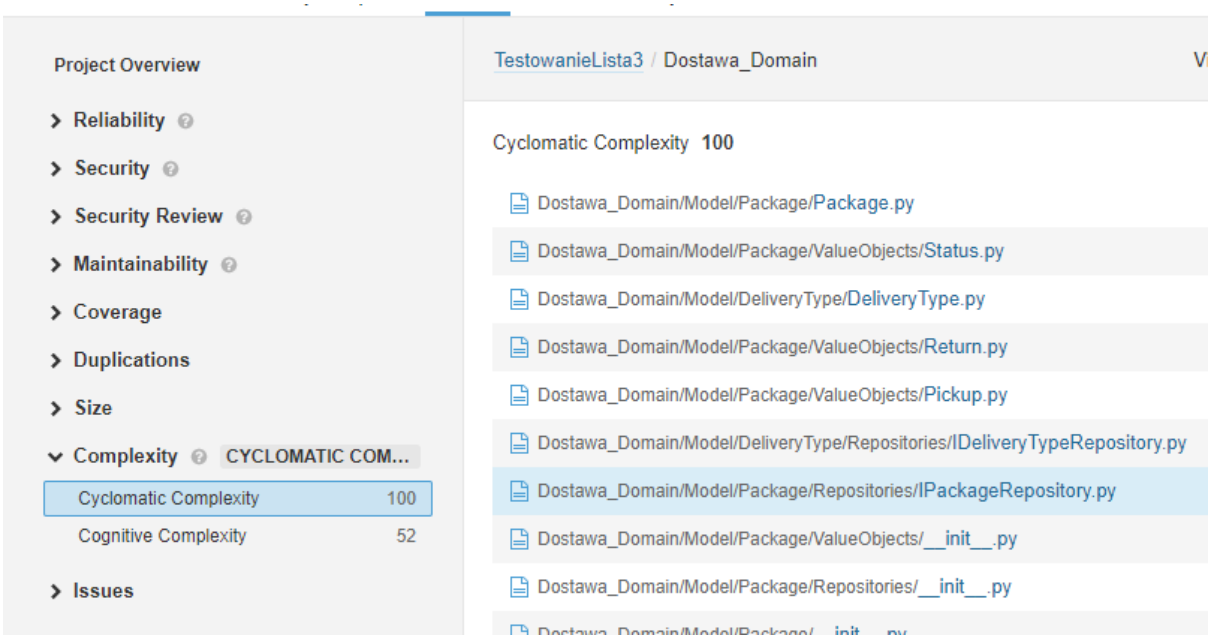
Programem przeanalizowanym przez SonarQube jest mój projekt z innego przedmiotu. Posiada ponad 1000 linijek kodu, więc jest dobrym przykładem do przetestowania SonarQube .
https://github.com/Konrad-Kasprzyk/study/tree/master/year%203/aplikacje%20z%20bazami%20danych/Dostawa/src
Po załadowaniu projektu i zakończonej analizie pokazuje się powyższy widok. Można z niego odczytać czy kod zawiera wycieki bezpieczeństwa, ile kodu się duplikuje, jaki procent kodu pokryty jest testami oraz ile złych praktyk programowania prawdopodobnie popełniliśmy. Przydatną rzeczą jest szacowanie czasochłonności naprawienia wszystkich niedoskonałości naszego kodu.



Przeglądając wykryte usterki w kodzie, mamy od razu oznaczony poziom zagrożenia jaki wykrył program. Dzięki temu możemy od razu przejść do najważniejszych problemów z kodem.



Wykryte błędy w kodzie możemy skomentować, przypisać danej osobie, podejrzeć kto dany kod dodał. Mamy dowolność w zarządzaniu wykrytymi błędami. Możemy zmienić ich priorytet lub nawet usunąć oznakowanie że dany kawałek kodu jest błędem.



Metryki kodu stosowane przez SonarQube są uporządkowane i opisane. Łatwo zobaczyć te części kodu, które łamią tylko wybraną metodykę. Każde złamanie danej metodyki jest opisane. Podany jest powód dlaczego powinno się zmienić kod i przykład jak powinno się poprawnie zastosować daną metodykę.

Zadanie 3

Nietrywialną metryką dostarczaną przez CCCC jest z pewnością McCabe's Cyclomatic Complexity (MVG), a więc złożoność cyklomatyczna. Jej formalną definicją jest liczba liniowo niezależnych ścieżek przepływu w programie. Przybliżeniem tej liczby może być szukanie słów kluczowych i operatorów w kodzie źródłowym, które mogą powodować powstanie nowych drzewek decyzyjnych. W przypadku języka C++ wartość jest zwiększana w przypadku napotkania któregoś z tych tokenów: `if`, `while`, `switch`, `break`, `for`, `&&`, `||`. Jak wspomniałem wyżej w przypadku analizy z użyciem CCCC, MVG ma wyznaczone widełki mówiące o tym, jak bardzo złożony jest kod i czy może powodować duże ryzyko.

SonarQube

1. Functions and methods should not be empty

Nie chodzi o to że puste metody czy klasy są bezużyteczne. One mogą mieć bardzo duże znaczenie w pewnym kontekście. Obiekt pustej klasy lub sama klasa może być użyta w wielu wzorcach programowania. Możliwe że funkcja nic nie wnosi, ale musi wystąpić np. dla zaimplementowania interfejsu. Jednak, gdy dla autora takiego kodu jest to oczywiste. To dla nowej osoby, nie znającej całego projektu i intencji autora może to być duży problem w zrozumieniu kodu. Osoba może źle pomyśleć że funkcja lub klasa jest niedokończona, lub że powinna pusta. Powinno się w tym miejscu napisać krótki opis dlaczego tak kod wygląda lub rzucić wyjątkiem sygnalizujący brak implementacji.

2. Shadowing a builtin

Jedną z metodyk jest sprawdzanie, czy zadeklarowane nazwy zmiennych i funkcji nie ukrywają wbudowanych słów kluczowych lub funkcji. Ukrywanie wbudowanych słów języka powoduje trudniejszy kod w utrzymaniu. Kod ciężej się czyta i może być powodem błędów, gdy ktoś nie zauważy, że jest zasłonięte słowo kluczowe. Szczególne jeśli ktoś po raz pierwszy dany kod czyta.

3. Naming convention

Dobre nazewnictwo funkcji, metod, klas, zmiennych różnego typu jest bardzo ważna. Przekłada się to na jakość czytania kodu. Czytający, kiedy napotyka ogólnie stosowane konwencje, od razu wie co dany kod robi, nawet nie sprawdzając jakiego typu jest to metoda lub zmienna. Np. dla języka Python istnieją oficjalne normy pisania kodu - PEP 8 i PEP 423.