

Zasady tworzenia API dla grupy Invenzzia

Tomasz Jędrzejewski

28 stycznia 2011

Niniejszy dokument prezentuje zasady tworzenia i projektowania API dla projektów open-source opracowywanych przez grupę Invenzzia. Aby projekt mógł być wydany pod patronatem grupy, musi stosować się do nich. W przypadku projektów archiwalnych, nowe wersje powinny natomiast dążyć do uzyskania zgodności z tym dokumentem.

Dokument koncentruje się na przedstawieniu dobrych i złych praktyk programistycznych, wraz z przedstawieniem konsekwencji ich stosowania. Na początku rozważane są elementy języka PHP oraz zasady ich poprawnego stosowania w projektach. Później przechodzimy do omówienia technik i analizy rozwiązań, by zakończyć na zastosowaniach wzorców projektowych. Dodatkowy rozdział poświęcony jest technikom automatycznego ładowania.

Spis treści

1	Główne koncepcje i założenia	2
1.1	Magia	2
1.2	Efekty uboczne	3
1.3	Argumenty funkcji	4
1.4	Elementy statyczne	5
1.5	Wyjątki	5
1.6	Kompozycja nad dziedziczenie	6
1.7	POPO	6
1.8	Mechanizmy nieobiektywne	6
2	Automatyczne ładowanie klas	7
2.1	PSR-0	7
2.2	Dodatkowe założenia	8
3	Rozwiązania typowych problemów	8
3.1	Ładowanie danych zewnętrznych	8
3.2	Obsługa konfiguracji	9
3.3	Sygnalizacja braku elementu	9
3.4	Przekazanie do wyjątku dodatkowych informacji	9
3.5	Aplikacje konsolowe	10
4	Wzorce projektowe	10
4.1	Stosowanie wzorców w projektach	10
4.2	Zalecane i niezalecane wzorce	11
4.3	Użycie Obserwatora	11

4.4	Katalog grupy Invenzzia	11
5	Wdrażanie zasad	12
6	Podsumowanie	12

1 Główne koncepcje i założenia

Głównymi założeniami projektowymi, jakie powinny przyświecać programiście podczas projektowania API są:

- lekkość interfejsu,
- łatwość rozbudowy interfejsu,
- unikanie rozwiązań magicznych,
- unikanie narzucania określonych rozwiązań bez wyraźnego powodu,
- znajomość efektów ubocznych tworzonego kodu i umiejętność ich udokumentowania.

1.1 Magia

Elementami magicznymi języka PHP nazwiemy wszystkie konstrukcje językowe, które udają, że są czymś innym niż w rzeczywistości. Działaniami magicznymi nazwiemy wszelkie działania, które wykonują w tle czynności, których programista ma prawo się nie spodziewać i odwołują się do danych, do których z punktu widzenia użytkownika w danym miejscu nie powinno być dostępu.

Powyższa definicja obejmuje większość tzw. magicznych metod takich, jak `__call()`, `__get()` czy `__set()` oraz elementy statyczne klas, które w większości przypadków prowadzą właśnie do powstania magii. Jednak działania magiczne mogą być wywołane także przez pozornie bezpieczne konstrukcje językowe.

Stosowanie metod magicznych w projekcie API prowadzi do zaciemnienia obrazu tego, jak ono właściwie działa. Programista, wywołując metodę czy odwołując się do pola, właściwie nie ma pojęcia o tym, co właśnie wywołał. Wszystko jest w porządku, dopóki kod działa, lecz gdy przestanie, utrudnia to poważnie diagnozę tego, co się właściwie wydarzyło. Innym argumentem przeciwko jej stosowaniu jest brak dobrego wsparcia ze strony edytorów kodu oraz dokumentacji. Edytory nie są w stanie poprawnie śledzić magicznych odwołań, ponieważ z ich punktu widzenia dana metoda czy pole po prostu nie istnieje. Analogiczny problem napotyka programista, który próbuje znaleźć w dokumentacji informacje o metodzie `xyz()` tworzonej w sposób magiczny — jej nazwa przetwarzana jest dynamicznie, zatem jest bardzo mała szansa, że ta konkretna nazwa będzie udokumentowana i będzie możliwe znalezienie jej poprzez wyszukiwarke.

Elementy magiczne mogą także wprowadzać sztuczne ograniczenia projektowe. Wyobraźmy sobie hipotetyczny ORM, który w sposób magiczny udostępnia metody w stylu `getUsersByName()`. Dopóki tabela oraz nazwa pola jest zakodowana na sztywno, taki zapis jest czytelny i poprawny, lecz co w sytuacji, gdy któryś z tych elementów chcemy konfigurować dynamicznie, ładując go np. ze zmiennej? Programowanie obiektowe przewiduje, że takie dane powinny być

przekazywane jako argumenty, tymczasem tutaj zmuszeni jesteśmy obchodzić sztuczne ograniczenie:

```
$foo->__call('getUsersBy'.$fieldName, array(...));

$methodName = 'getUsersBy'.$fieldName;
$foo->$methodName(...);
```

Jako programiści musimy uświadomić sobie, że tego typu metody magiczne nie są nam do niczego potrzebne z technicznego punktu widzenia. Powyższy przypadek można łatwo naprawić, udostępniając metodę `getDataBy()`, która pobiera nazwę tabeli oraz pola jako najzwyklejsze argumenty. Jeśli obiekt musi przechowywać zmienną liczbę atrybutów, zamiast metod `__get()` i `__set()` powinniśmy zastosować jawnie nazwane metody w stylu `getAttribute()` czy `setAttribute()`.

1.2 Efekty uboczne

Funkcja lub metoda posiada *efekt uboczny*, jeśli oprócz wyprodukowania wyniku, zmienia stan programu lub wykonuje pewną interakcję ze światem zewnętrznym. Wbrew swej nazwie, niekiedy efekt uboczny jest pożądanym działaniem określonej funkcji, np. w operacjach wejścia/wyjścia. Efekty uboczne mogą jednak utrudniać wykorzystywanie tworzonego systemu, zwłaszcza gdy nie są śledzone i dokumentowane.

Problemy, na jakie możemy natknąć się, nadużywając efektów ubocznych:

- funkcja może przypadkowo zmodyfikować stan innego komponentu, prowadząc do awarii,
- analiza kodu z dużą liczbą źle zdefiniowanych efektów ubocznych jest bardzo ciężka,
- duża liczba efektów ubocznych może sprawić, że ciężko będzie używać większej liczby takich funkcji jednocześnie, a kod niezbędny do radzenia sobie z nimi będzie rzutować na wydajność,
- efekty uboczne utrudniają testowanie aplikacji.

W projektach grupy Invenzzia nie chodzi o to, by efektów ubocznych nie stosować, ale by miały one ograniczony i dobrze zdefiniowany zasięg. Rozważmy hipotetyczną metodę `foo()`, która przyjmuje za argument jeden obiekt. Jako generalną regułę można przyjąć następujące wymagania:

- `foo()` może zmienić stan obiektu, na którym ją wywołujemy,
- `foo()` może zmienić stan obiektów zależnych połączonych relacją kompozycji z obiektem, na którym ją wywołujemy,
- `foo()` może zmienić stan obiektów zależnych połączonych słabszymi relacjami tylko wtedy, gdy explicite wynika to z nazwy oraz definicji tej metody,
- `foo()` może zmienić stan obiektu przekazanego za argument tylko wtedy, gdy explicite wynika to z jej nazwy i definicji.

Ponadto, `foo()` może korzystać jedynie ze stanu tych obiektów, które zostały jawnie wstrzyknięte do wywołującego obiektu.

Efekty uboczne metod i funkcji muszą być udokumentowane.

1.3 Argumenty funkcji

Zasady użycia argumentów funkcji i metod są następujące:

- nazwy zmiennych argumentowych muszą być czytelne i zrozumiałe,
- dopuszczalne jest stosowanie argumentów opcjonalnych, z predefiniowaną wartością,
- nie jest dopuszczalne stosowanie nielimitowanej liczby argumentów. Liczba argumentów musi być z góry określona,
- unikamy stosowania długich list argumentów liczących wiele pozycji,
- jeśli funkcja musi pobierać zbiór logicznych przełączników przyjmujących wartości `true` lub `false`, wykorzystujemy flagi logiczne przekazywane jako pojedynczy argument,
- odradzane jest tworzenie funkcji, które mogą mieć dwa zestawy argumentów o innej semantyce (np. jeśli pierwszy argument jest ciągiem tekstowym, to należy podać też drugi, a jeśli jest tablicą, drugiego nie podajemy) — w takim przypadku oba warianty powinny zostać zaimplementowane jako osobne funkcje.

Zamiast wykorzystywania zmiennej liczby argumentów, funkcja bądź metoda powinna pobierać zwykłą tablicę.

Przyjmowane są następujące konwencje:

- metody, które pozwalają zarejestrować pewien obiekt pod jakimś kluczem będącym ciągiem tekstowym, powinny pobierać klucz jako pierwszy, zaś obiekt jako drugi argument.
- powinno się unikać przekazywania w jednym zestawie argumentów więcej niż jednego obiektu — punkt ten można pominąć, jeśli jasno wynika to z semantyki funkcji i inne rozwiązania byłyby po prostu nieeleganckie,
- dane do przetwarzania przez metodę powinny być przekazywane jako pierwszy argument, a dopiero potem dodatkowe atrybuty konfiguracyjne samo przetwarzanie.

Wyjaśnienia wymaga ostatnia konwencja. Takie dane umieszcza się albo na końcu, albo na początku listy argumentów. Oba sposoby mają swoje wady i zalety. Jeśli są umieszczone na końcu, tworzenie argumentów opcjonalnych jest mocno utrudnione; możemy albo dopuścić, by w szczególnych przypadkach przetwarzane dane były w środku listy argumentów (jak to czasem ma miejsce w funkcjach PHP), albo zrobić wyjątek i przenieść je na początek. Z drugiej strony, jeśli są one wszędzie przekazywane jako pierwszy argument, zagnieżdżanie funkcji sprawia, że pozostałe argumenty będziemy musieli podawać w kodzie kilka linii niżej i trzeba będzie zliczać nawiasy bądź zagnieżdżenia, by odkryć, której funkcji one dotyczą. Doszliśmy do wniosku, że jednolita konwencja w zakresie podawania argumentów jest ważniejsza, niż zagnieżdżanie, ponieważ wprowadza mniej ograniczeń oraz redukuje liczbę pomyłek.

1.4 Elementy statyczne

Zabronione jest używanie w klasach statycznych pól, zarówno prywatnych, jak i publicznych. Wprowadzają one trudny do testowania i kontrolowania stan globalny, który z uwagi na nieprzewidywalne efekty uboczne nie może być zaakceptowany.

Metody statyczne są dopuszczalne pod warunkiem, że nie mają efektów ubocznych, lub ograniczają się one do obiektu, na którym taka metoda jest wywoływana. Jednak w tym drugim przypadku zalecane jest, aby statyczna metoda była po prostu zwykłą metodą danej klasy, skoro już operuje na obiekcie. Metody statyczne powinny być stosowane do udostępniania jakichś operacji na danych, które nie wymagają obecności obiektu tej klasy. Przykładem może być liczenie slugów w modelach. Operacja jest uniwersalna i może być wykorzystywana w wielu miejscach, a dodatkowo nie ma efektów ubocznych (obliczony slug zależy wyłącznie od tekstu podanego za argument), więc jest dobrym kandydatem na bycie metodą statyczną.

1.5 Wyjątki

Do raportowania nieprawidłowości musi być używany mechanizm wyjątków dostępny w PHP. Projekty nie powinny tworzyć własnej hierarchii klas wyjątków, lecz bazować na tej z rozszerzenia SPL (*Standard PHP Library*). Definiuje ono dwie ogólne i kilka bardziej szczegółowych klas wyjątków tworzących załazek spójnej hierarchii.

Dwa podstawowe wyjątki to **RuntimeException** oraz **LogicException**. Pierwsza z nich reprezentuje błędy, które można wychwycić jedynie podczas działania aplikacji i wynikają one z nieprawidłowego korzystania z aplikacji przez użytkownika. **LogicException** reprezentuje natomiast błędy wynikające z próby błędnego użycia elementu poprzez podanie np. niepoprawnych z jego punktu widzenia danych konfiguracyjnych. Przykładem może być tu problem sprawdzania długości pliku. Jeśli stwierdzimy, że załadowany plik nie ma ustalonej przez nas długości, powinniśmy rzucić błąd wywodzący się od **RuntimeException**, natomiast jeśli do metody sprawdzającej długość podamy, że długość pliku ma być ujemna, wtedy prawidłowym wyjątkiem jest **LogicException**, gdyż nie ma czegoś takiego, jak „ujemna długość”.

Dokładny opis klasy **LogicException** mówi: *wyjątek powinien być rzucony, gdy pewne wyrażenie logiczne jest nieprawidłowe*. Zwrot „nieprawidłowe” oznacza tu błędne sformułowanie wyrażenia, a nie wartość logiczną, jaką ono przyjmuje. Nieprawidłowe wyrażenia nie mogą zwracać wartości logicznej, ponieważ są nieprawidłowe i nie mają sensu logicznego.

Od tych dwóch klas wyprowadzonych zostało kilka bardziej szczegółowych wyjątków, które można użyć w konkretnych sytuacjach, pamiętając jednak o znaczeniu klasy bazowej. Oznacza to, że np. **LengthException** nie możemy rzucić w sytuacji, gdy plik ma nieprawidłową długość, ponieważ dziedziczy ona po **LogicException**, która oznacza coś innego.

Dopuszczalne, a wręcz wskazane jest tworzenie własnych klas wyjątków na bazie hierarchii klas SPL. Domyślny zestaw klas nie pokrywa wszystkich przypadków, dlatego najprawdopodobniej w naszych projektach będziemy musieli stworzyć własne, np. **AuthException**. Powinny być one składowane w przestrzeni nazw projektu **Foo\Exception** i nosić nazwy w stylu **BarException**.

Do klas można także dodawać nowe metody i pola tak, aby obiekty wyjątków mogły przenosić dodatkowe informacje.

1.6 Kompozycja nad dziedziczenie

Dziedziczenie jest podstawowym mechanizmem rozszerzania implementacji istniejącego komponentu bądź klasy o nową funkcjonalność. W PHP, podobnie jak w wielu innych językach, dopuszczalne jest wyłącznie dziedziczenie jednokrotne, w którym klasa może dziedziczyć po co najwyżej jednej klasie bazowej. Rodzi to pewne ograniczenia, lecz z drugiej strony mechanizm wielokrotnego dziedziczenia prowadzi do wielu patologii związanych z potencjalnymi konfliktami metod.

W projektach grupy Invenzzia obowiązuje zasada *przedkładania kompozycji nad dziedziczenie*, a także będąca pewną podbudówką dla niej *zasada jednej odpowiedzialności*. Mówi ona, że nigdy nie powinno być więcej niż jednego powodu do modyfikacji klasy. Przez powód do modyfikacji rozumiemy odpowiedzialność, czyli pewną rzecz, którą klasa się zajmuje. Stosowanie tej zasady prowadzi do powstania większej ilości mniejszych, ale bardziej wyspecjalizowanych klas komunikujących się ze sobą poprzez publiczne interfejsy. Rozbudowa lub naprawa błędów ma bardziej lokalny zasięg, a dzięki temu tworzone oprogramowanie jest łatwiejsze w konserwacji i utrzymaniu.

Kompozycja to proces, w którym funkcjonalność obiektu jest komponowana dynamicznie z szeregu mniejszych obiektów. Oferuje większą elastyczność, niż dziedziczenie, ponieważ zestaw obiektów możemy w dowolnym momencie przebudować oraz przekonfigurować bez modyfikacji kodu źródłowego. Ponadto, nie dotyczą jej ani ograniczenia jednokrotnego, ani patologie wielokrotnego dziedziczenia.

1.7 POPO

Skrót POPO pochodzi od *Plain Old PHP Objects*. W świecie programowania PHP oznacza on, że gdy użytkownik musi stworzyć pewien własny kod w oparciu o mechanizmy naszej biblioteki, nie zmuszamy go do kłopotliwego dziedziczenia lub implementowania niepotrzebnych interfejsów. Dzięki temu może on wykorzystać do swoich celów dowolną klasę o dowolnym interfejsie. Możliwość stosowania POPO redukuje liczbę zależności i ułatwia integrację z innymi bibliotekami.

1.8 Mechanizmy nieobiektywne

Zasady użycia mechanizmów nieobiektywnych języka PHP:

- zakładamy, że opcja `magic_quotes` jest wyłączona,
- w przypadku tablic superglobalnych, korzystamy wyłącznie ze skróconych form: `$_SERVER`, `$_POST` itd. a ich użycie, ze względu na obecność stanu globalnego musi być dokładnie udokumentowane.

Rozszerzenia PHP 5.3, których obecność możemy przyjąć za pewnik:

- Filter,

- GD,
- Intl,
- PCRE Regular Expressions,
- PDO,
- PHAR,
- Zlib

Dodatkowo, dopuszczalne jest korzystanie z mechanizmów pamięci współdzielonej poprzez jednolity interfejs cache'ujący tak, aby była możliwość obsługi więcej niż jednego backendu.

2 Automatyczne ładowanie klas

Kod tworzonych bibliotek musi być dostosowany do prawidłowego działania w trybie automatycznego ładowania klas. Jako standard nazewnictwa oraz tłumaczenia nazw przyjmujemy *PSR-0 Proposal* stworzoną przez *PHP Standards Working Group*. Umożliwi to prawidłową współpracę tworzonych przez grupę Invenzzia kodu z bibliotekami innych zespołów używających tego samego standardu. Można do nich zaliczyć projekty *Doctrine 2*, *Symfony 2* oraz *Zend Framework 2*.

2.1 PSR-0

Założenia standardu:

1. W pełni kwalifikowana nazwa klasy wraz z przestrzenią nazw ma postać `\<Dostawca>\(<Przestrzeń nazw>\)*Nazwa klasy`.
2. Każda przestrzeń nazw musi posiadać przestrzeń nazw głównego poziomu (tzw. nazwa dostawcy)
3. Każda przestrzeń nazw może posiadać dowolną liczbę podprzestrzeni i poziomów zagnieżdżeń.
4. Każdy separator przestrzeni nazw jest zamieniany na `DIRECTORY_SEPARATOR` podczas ładowania klasy z systemu plików.
5. Każde podkreślenie `_` w nazwie klasy jest zamieniane na `DIRECTORY_SEPARATOR` podczas ładowania klasy z systemu plików.
6. Podkreślenia `_` nie mają żadnego specjalnego znaczenia w nazwach przestrzeni nazw.
7. W pełni kwalifikowana przestrzeń nazw oraz nazwa klasy posiada przyrostek `.php` podczas jej ładowania z systemu plików.
8. W nazwach klas, przestrzeni nazw i dostawców można używać zarówno dużych, jak i małych liter alfabetu.

2.2 Dodatkowe założenia

Dodatkowe założenia dla projektów grupy Invenzzia:

1. Nazwy klas, przestrzeni nazw i dostawców muszą zaczynać się dużą literą.
2. Unikamy zagnieżdżeń powyżej pięciu poziomów.
3. Położenie klas w systemie plików wyznacza wyłącznie nazwa dostawcy. Nie można zmienić lokalizacji podrzędnych przestrzeni nazw.
4. W nazwach klas oraz przestrzeniach nazw nie powinny występować podkreślenia `_`.
5. Nie wolno używać `require`, `include`, ani podobnych instrukcji do doładowywania klas zależnych w nagłówkach pliku.
6. Ręczne ładowanie klas przy pomocy `require` oraz `include` (np. w celu sprawdzenia czy taka klasa w ogóle istnieje w systemie) może być wykonane tylko wtedy, jeśli jest ono kompatybilne z ładowaniem automatycznym. Innymi słowy, taka klasa musi być też poprawnie ładowana przez autoloader. Do uzyskania ścieżki do głównego elementu przestrzeni nazw powinna być użyta refleksja.
7. Pliki niepodlegające automatycznemu ładowaniu (np. konfiguracja, proste skrypty PHP niezawierające klas) nie powinny być wymieszane z resztą kodu.
8. Klasy abstrakcyjne powinny mieć nazwę w stylu **AbstractFoo**.
9. Dla interfejsów dopuszczalne są dwie konwencje: **FooInterface** lub forma przymiotnikowa czasownika, np. **Countable**, **Readable**. Drugi wariant powinien być stosowany w przypadku większych rodzin interfejsów ogólnego przeznaczenia.

Powyższe reguły mają charakter wewnętrzny i nie kolidują z konwencjami stosowanymi w innych projektach. Żelazna zasada brzmi: *kod projektów Invenzzia musi być ładowalny przy użyciu dowolnego autoloadera zgodnego z PSR-0*.

3 Rozwiązania typowych problemów

W sekcji tej opisywane są typowe problemy projektowe i konwencje ich rozwiązywania w obrębie projektów grupy Invenzzia.

3.1 Ładowanie danych zewnętrznych

Ładowanie danych zewnętrznych powinno być obsługiwane przez tzw. *ładowarki*, analogicznie do rozwiązania z Symfony 2. Komponent, który chce wczytać z zewnętrznego źródła jakieś dane, powinien udostępnić interfejs dla ładowarki, który następnie jest rozszerzany przez konkretne klasy implementujące różne formaty. Dla klas czytających dane z pliku powinna być zdefiniowana jednolita klasa bazowa obsługująca operacje na pliku, definiowanie ścieżek oraz sprawdzanie czy plik istnieje.

Konwencje nazewnictwa:

- Nazwa interfejsu ładującego: **LoaderInterface** w przestrzeni nazw **Loader** danego bytu.
- Nazwa klasy ładującej: **BarLoader** w przestrzeni nazw **Loader** danego bytu.
- Nazwa klasy bazowej dla plikowych ładowarek: **FileLoader**.

3.2 Obsługa konfiguracji

Obszerniejsza konfiguracja powinna być wstrzykiwana do obiektów danego komponentu jako samodzielny obiekt przeznaczony tylko do tego celu. Obiekt ten może być bytem ogólnym wielokrotnego przeznaczenia, jak i bytem dedykowanym do rozwiązania konkretnego problemu, bez możliwości użycia gdzie indziej. Klasy z małą liczbą opcji mogą być konfigurowane bezpośrednio poprzez konstruktor, gettery i settery.

3.3 Sygnalizacja braku elementu

Wiele tworzonych klas przechowuje zbiory rozmaitych wartości, dostarczając interfejs do ich ustawiania i pobierania. Jeśli żądany element nie istnieje, możemy zareagować następująco:

- rzucić wyjątek — rozwiązanie dla sytuacji, gdy zakładamy, że system powinien odwoływać się wyłącznie do istniejących elementów.
- zwrócić `null` — rozwiązanie dla sytuacji, gdy dopuszczamy możliwość nieistnienia elementów.

Istotne jest, że musi to być wartość `null`, w przeciwieństwie do konwencji PHP zakładającej zwracanie wtedy `false`, która jednak jest niejednoznaczna. `null` powinien być używany zawsze, gdy zakładane jest zwrócenie jakiegoś elementu i z jakiegoś (niekrytycznego) powodu się to nie udało. Wartość `false` traktujemy jako poprawną wartość logiczną i możemy ją stosować jedynie wtedy, gdy metoda zwraca wyłącznie wartości logiczne.

3.4 Przekazanie do wyjątku dodatkowych informacji

Rozważmy projekt *Open Power Template*, w którym działał złożony, rozciągający się na wiele komponentów kompilator. Komponenty te mogły rzucać wyjątki w celu poinformowania o błędzie kompilacji, jednak aby były one przydatne dla użytkownika końcowego, przy ich rzuceniu należało wypełnić je dodatkowymi informacjami takimi, jak np. nazwa szablonu, w którym błąd się pojawił. Pobieranie i wpisywanie do wyjątku tych danych podczas jego rzucania mogło mocno komplikować kod, dlatego zastosowane zostało tam inne rozwiązanie. Cały algorytm kompilacji miał jeden punkt wejścia, którym była metoda `compile()`. Zawierała ona blok `try ... catch`, który przechwytywał wszystkie wyjątki z kompilatora, dodawał do nich niezbędne informacje i przekazywał dalej:

```
try
{
    // algorytm
```

```

}
catch(SomeException $exception)
{
    $this->populateException($exception);
    throw $exception;
}

```

3.5 Aplikacje konsolowe

Niebawem pojawi się tu opis wytycznych dotyczących tworzenia aplikacji narzędziowych obsługiwanych z poziomu konsoli. Nie dotyczą one interfejsów wiersza poleceń do frameworków czy aplikacji WWW, lecz narzędzi dedykowanych do pracy w konsoli takich, jak np. *TypeFriendly*.

4 Wzorce projektowe

W tej sekcji skupimy się na zagadnieniu implementowania wzorców projektowych w projektach grupy Invenzzia. Pierwszą podstawową zasadą brzmi: *nie umieszczamy wzorców projektowych w kodzie na siłę, byle jakieś były*. Projekt, w którym nie ma ani jednego wzorca jest dużo lepszy, niż projekt w którym jest ich 5, ale użytych bez sensu. Podstawową przesłanką do podjęcia decyzji o użyciu wzorca musi być napotkanie jakiegoś uzasadnionego potrzebami problemu projektowego. Przykładowo, ideą działania systemu może być konieczność jego rozbudowy przez użytkownika w miejscu A, co wymusza pewną elastyczność w miejscu B. Miejsce B musi być rozszerzalne, ponieważ wynika to z wymagań projektu, zatem kolejnym krokiem będzie zastanowienie się, jakiego rodzaju elastyczności potrzebujemy i sprawdzenie czy jakiś wzorec projektowy nie spełnia naszych oczekiwań.

Kontrprzykładem może być sytuacja, w której stwierdzamy: „mam już pewien interfejs, to dodamy do niego wzorec *Odwiedzający*, bo może się komuś przydać”. Zastosowanie wzorca nie jest tu niczym uzasadnione. Użyty argument nie jest żadnym argumentem, ponieważ nie wynika z założeń projektowych, ani z wymagań, lecz jest zwykłym kaprysem. Jeśli ktoś faktycznie będzie potrzebować w tym miejscu wspomnianego wzorca, powinien go sobie sam zaimplementować, a dobrze zaprojektowane API nie powinno mu tego utrudniać.

Przed implementacją wzorca należy przejrzeć konsekwencje stosowania i zastanowić się czy są one przez nas akceptowalne. Część wzorców wprowadza określone możliwości kosztem innych, dlatego kluczowa jest umiejętność stwierdzenia czy w ogóle możemy coś poświęcić. Jeśli żaden ze wzorców nie spełnia naszych oczekiwań, po prostu projektujemy coś własnego. Wzorce są zwykłą skrzynką narzędziową, która nie jest remedium na wszystkie problemy i piszą o tym w swojej książce sami ich twórcy[1, s. 16, 342].

4.1 Stosowanie wzorców w projektach

Każdy użyty przez nas wzorec projektowy musi być jawny. Powinniśmy oznaczyć w kodzie, że korzystamy ze wzorca X zdefiniowanego w katalogu Y. Analogiczny zapis powinien znaleźć się również w dokumentacji programistycznej. Jeśli używamy zmodyfikowanego wariantu wzorca, konieczne jest zaznacze-

nie tego faktu poprzez dodanie do nazwy słowa *modified* oraz opisanie zakresu zmian, wraz z ich konsekwencjami.

Wzorce mogą pochodzić z różnych katalogów. Podstawowym jest katalog [1], który wprowadził to pojęcie do informatyki i zdefiniował pierwsze 23 wzorce. W dokumentacji powinno się określać go mianem *GoF book*.

4.2 Zalecane i niezalecane wzorce

Wzorce projektowe, które muszą być bezwzględnie znane przez programistów grupy Invenzzia to:

- *Dekorator*, GoF,
- *Fabryka abstrakcyjna*, GoF,
- *Iterator*, GoF,
- *Metoda fabrykująca*, GoF,
- *Obserwator*, GoF,
- *Odwiedzający*, GoF,
- *Strategia*, GoF.

Zabronione jest wykorzystywanie w projektach wzorca *Singleton*, który na potrzeby grupy Invenzzia traktowany będzie jako antywzorec, ponieważ korzysta z zakazanych pól statycznych. Projekty nie mogą zakładać ograniczania liczby obiektów — użytkownik ma prawo samodzielnie zdecydować, ile obiektów potrzebuje i kiedy powinny być one niszczone. Można co najwyżej dać zalecenie odnośnie ilości obiektów, które wystarczą do optymalnej pracy z systemem.

4.3 Użycie Obserwatora

Wzorec *Obserwator* nie powinien być implementowany od zera. Rozszerzenie SPL dostarcza gotowych interfejsów zarówno dla obiektu obserwowanego, jak i obserwatora:

- `SplObserver`[2]
- `SplSubject`[3]

Projekty powinny korzystać z tych interfejsów do implementowania tego wzorca.

4.4 Katalog grupy Invenzzia

Dokument ten powołuje do istnienia także katalog wzorców projektowych grupy Invenzzia, który powinien być rozbudowywany o dobre rozwiązania zastosowane w zrealizowanych już projektach. Propozycję wzorca może zgłosić każdy projektant API, lecz aby została ona ostatecznie przyjęta, musi zostać oceniona przez innych programistów, którzy mają prawo zgłaszania swoich własnych poprawek i rozwinięć. Decyzję o wpisaniu propozycji do katalogu musi

podjąć trzech programistów. Szczególny nacisk musi być położony na sprawdzenie czy zgłaszana propozycja nie jest w rzeczywistości antywzorcem oraz czy nie korzysta z takowych.

Katalog może zawierać wzorce także z innych dziedzin projektowych i każda z nich najprawdopodobniej będzie posiadać własne wytyczne odnośnie elementów opisu konkretnego wzorca. Do wzorców obiektowych na pewno można przyjąć konwencje zastosowane w [1].

5 Wdrażanie zasad

Projekty archiwalne i odziedziczone nie muszą spełniać opisanych w tym dokumencie wytycznych, ale w przypadku tworzenia nowej wersji ich API powinno zostać przeprojektowane tak, by uwzględniło niniejsze reguły.

Do pobrania są specjalne listy kontrolne, które mogą posłużyć jako pomoc podczas weryfikacji interfejsu programistycznego. Mają one formę pytań dotyczących badanego komponentu, na które odpowiadamy *Tak* lub *Nie*. Interfejs możemy uznać za zgodny z tym dokumentem, jeśli na wszystkie pytania odpowiedzieliśmy *Tak*.

Spis dostępnych list kontrolnych:

1. Lista kontrolna dla systemu,
2. Lista kontrolna dla komponentu,
3. Lista kontrolna dla klasy,
4. Lista kontrolna dla wzorca projektowego,
5. Lista kontrolna dla metody.

6 Podsumowanie

Obecna wersja dokumentu zawiera większość z planowanej zawartości. Fundamentalne zmiany założeń nie są przewidywane, natomiast niektóre kwestie mogą zostać w przyszłości doprecyzowane. Do napisania wciąż pozostał rozdział z zaleceniami dla aplikacji konsolowych. Do dokumentu mogą zostać dodane zupełnie nowe sekcje tak, aby uwzględnić rozwój języka PHP. Będą one koncentrować się na nowowprowadzanych elementach.

Literatura

- [1] E. Gamma, R. Helm, R. Johnson, J. Vlissides, „*Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku*”, wyd. Helion 2010, ISBN 978-83-246-2662-5
- [2] <http://docs.php.net/manual/en/class.spobserver.php>
- [3] <http://docs.php.net/manual/en/class.splsubject.php>