Poradnik programisty

Orodlin Team

wersja 0.2

sierpień 2007

Spis treści

1 Wstępniak – wstrętniak

Wiadomo, że nikt tego nie czyta, ale coś skrobnąć trzeba. Niniejsze opracowanie ma pomóc wypracować i stosować pewien standard efektywnego pisania kodu na potrzeby gry Orodlin i silnika Vallheru. Nie pojadłem wszyskich rozumów, ale znam już parę sztuczek, dzięki którym powstaje kod lepszy od tego zostawionego przez Thindila. Zachęcam do lektury, pytania, stosowania i zgłaszania uwag (i całych rozdziałów ☺), dzięki którym dokument będzie się rozwijał.

Szanowny przyszły adminie „gry opartej na silniku Vallheru”!

Poradnik, podobnie jak silnik gry, będzie udostępniany publicznie w wersji opóźnionej i/lub upośledzonej w stosunku do tego, czym dysponuje Orodlin Team.

Do zrozumienia przeważającej części poradnika wymagane jest choćby szczątkowe doświadczenie z programowaniem w jakimkolwiek języku wysokiego poziomu: PHP, C/C++/C#, Java, Pascal/Delphi itd. Kursów PHP na necie jest multum i nauka od podstaw nie jest moim celem.

Na chwilę obecną nie planujemy tworzenia jakichkolwiek nakładek upgrade'ujących np. bazę danych z wersji 1.3 do 1.4. Wersja rozwojowa to rozwojowa, wszystko będzie cacy w 2.0 ;) Przetłumaczenie poradnika na angielski również jest planowane na okolice wydania 2.0.

Pozostaje mi zaproszenie Cię do współpracy w rozwijaniu silnika i poradnika☺

Konwencja typograficzna

Rzeczy niepewne, do zbadania, plany na przyszłość itd. są na żółtym tle. Przykładowy kod – `Courier New, rozmiar 10` (albo podobna czcionka o stałej szerokości znaku). Rzeczy, które są troszkę zaawansowane i można ominąć przy pierwszym czytaniu danego rozdziału – na niebiesko. Na szaro zaznaczono odnośniki do skakania po dokumencie.

Interlinia półtora, z wyjątkiem kodu.

2 Do zrobienia

Siłą rzeczy na razie mam tu taką sieczkę ze wszystkiego, skojarzenia wpisywane na gorąco itd. Można traktować jako „manifest admina technicznego”.

OGÓLNE

- Opis struktury plików i katalogów: od czego są includes, class, cache, templates... Wymienienie ważniejszych - jak na wiki technicznej. Co jest co w config.php, jak stosować includes/security.php, gdzie jest funkcja error() czy integercheck. Do tego podział strony na 3 częsci – head, „mięsko“ i foot. Taki ogólny, wstępny tekst. Nasze pluginy smarty, JS...
- Opis konfiguracji gry i serwera – wersje PHP, MySQL, Apache, Smarty itd. error-reporting(E_ALL), bugtrack (tylko tyle „że jest“). Docelowo poradnik instalacji (L)AMP i czego tylko jeszcze będziemy potrzebować (PHP5, eAccelerator, mod_rewrite, OPT, adodblite, adodb C extension...).
- Obsługa Subversion (checkout, commit, update, Id) i Kompare/diff/WinMerge
- Doxygen/phpdocumentor – styl komentowania kodu, przykłady, transformacja.
- Obsługa bugtracka, rozwiązywanie przykładowych błędów (MoveNext, smarty „gra nie zadziałała poprawnie“, „undefined index“ i inne Notice. tabelka adodb_logsql, $db -> ErrorMsg(), $db -> debug = true;)
- PHPMyAdmin i podstawy MySQL w konkretnych zastosowaniach Oro
- SELECT – jak dużo można wykonać z SQL'a. Zaawansowany MySQL – sumowanie kolumn wiersza i calej tabeli, SUM, MAX, ORDER BY, GROUP BY, AS, count(*), HAVING

OPTYMALIZACJA ISTNIEJĄCEGO I PISANIE NOWEGO KODU:

- Wydzielanie wspólnych części kodu, reusability -includes/artisan, security
- security.php, integercheck, SQL injection -markety? $db->qstr, strip_tags etc.
- Cache adodb, opóźnianie zapytań (walka w strażnicy zapisuje na sam koniec)
- Cena select w pętli a pobranie tabeli i przerobienie samemu - obciete where w resetach (warehouse) kontra punkty ataku strażki. LEFT JOIN
- JS: uzycie sliderów, tabów, ajaxa
- zmienne sesyjne, zmienne zmienne

OTWARTE DYSKUSJE, WISHLISTA, PLANY:

- Switch czy elseif (ja kontra Eld), duży array czy switch (ja kontra Klaus) – dyskusyjne ale wsio rawno, byle spójnie w całym pliku
- inne langi, lista polonizmów w kodzie do wywalenia. Rozbudować includes/config.php żeby zawierał np. nazwy ras/klas? Pory resetów? Początkowe ceny i ilości surowców w MK?
- Cięcie playersa – jak się rozpisałem na forum Vall
- Analiza GZIP kompresji – potrzebny nam poziom 9?
- zmniejszenie objętosci kodu gry do 1 MB (pliki kat. głównego + class, includes, languages, quests, templates + skrypty własne smarty/JS jakie beda), zapytan z heada foota do 10. Stopniowa rezygnacja z includes, rozbudowa classes
- czy potrzebujemy adodb_logsql skoro bierzemy tylko czas i ilość zapytan? moze jakis licznik wstawic do ado i **uj? albo w drugą stronę: uwalone zapytania pchać do bugtracka do analizy
- duże projekty: klany, miasta, ekwipunek, obciążenie postaci, perki + strategie ataków, czary, AI potworów (python?)
- Gazeta: XML + XSLT + XSD + CSS (wykorzystać markup DocBook?) + RSS
- porady klausa (elymantea): object factory, mvp, sprytne funkcje odpalane jak nie ma __get, __set
- 3D www – Blender? Czy da się Javascriptem wykonać mapę, czy trzeba flasha albo Javy.
- Opcja zipa dla gracza ze ścieżką i obrazkami, emotkami, JS itd (jak skórki Ogame)
- Masakrowanie all plikow co max 6 miesiecy, a zwlaszcza generujacych max transfer/procek/pamiec
- filmik promujący grę, nagrany choćby w CamStudio
- https
- rysowanie w PHP, thumbnaile avatarów
- SPL (PHP5)

3 Zagadnienia ogólne

Na razie wstawię tu na szybko kilka pomocnych zapytań MySQL, bo Dellasowi „się skasowało”:

Wsad do Magazynu Królewskiego:

```
TRUNCATE TABLE `warehouse`;
INSERT INTO `warehouse` (`reset`, `mineral`, `sell`, `buy`, `cost`, `amount`)
VALUES
(1, 'copperore', 0, 0, 1.000, 1000000000),
(1, 'zincore', 0, 0, 1.000, 1000000000),
(1, 'tinore', 0, 0, 1.000, 1000000000),
(1, 'ironore', 0, 0, 1.000, 1000000000),
(1, 'copper', 0, 0, 1.000, 1000000000),
(1, 'bronze', 0, 0, 1.000, 1000000000),
(1, 'brass', 0, 0, 1.000, 1000000000),
(1, 'iron', 0, 0, 1.000, 1000000000),
(1, 'steel', 0, 0, 1.000, 1000000000),
(1, 'coal', 0, 0, 1.000, 1000000000),
(1, 'adamantium', 0, 0, 1.000, 1000000000),
(1, 'meteor', 0, 0, 1.000, 1000000000),
(1, 'crystal', 0, 0, 1.000, 1000000000),
(1, 'pine', 0, 0, 1.000, 1000000000),
(1, 'hazel', 0, 0, 1.000, 1000000000),
(1, 'yew', 0, 0, 1.000, 1000000000),
(1, 'elm', 0, 0, 1.000, 1000000000),
(1, 'mithril', 0, 0, 1.000, 1000000000),
(1, 'illani', 0, 0, 1.000, 1000000000),
(1, 'illanias', 0, 0, 1.000, 1000000000),
(1, 'nutari', 0, 0, 1.000, 1000000000),
(1, 'dynallca', 0, 0, 1.000, 1000000000),
(1, 'illani_seeds', 0, 0, 1.000, 1000000000),
(1, 'illanias_seeds', 0, 0, 1.000, 1000000000),
(1, 'nutari_seeds', 0, 0, 1.000, 1000000000),
(1, 'dynallca_seeds', 0, 0, 1.000, 1000000000);
```

Ręczne dodanie gracza:

```
INSERT INTO players (`user`, `email`, `pass`) VALUES('Tester', 'test',
MD5('haslo'));
```

Dodaniesobie kasy i ustawienie mithrilu:

```
update players set credits=credits+10000, platinum=10 where id=1;
```

Energia:

```
update players set energy=100 where id=1;
```

Ręczne przenosiny do innej lokacji (Góry, Las, Lochy, Podróż...):

```
update players set miejsce='Altara' where id=1;
```

Statystyki, analogicznie:

poziom – `level`

staty: `strength wytrz inteli wisdom szyb agility`

umiejki: `atak shoot magia unik leadership`

`ability fletcher alchemia herbalist jeweller breeding mining lumberjack`

4 Konwencja kodowania

Poniższy tekst jest rozbudowaną wersją standardów kodowania ustalonych przez Thindila (ściślej punktów 1-9, bo potem ja wsadziłem 3 grosze do tekstu).

4.1 Konfiguracja edytora

Zalecany edytor: pod Microsoft®Windows™ całkiem fajny jest Notepad++ (TAK, używanie durnego zwykłego Notatnika będzie karane), pod GNU/Linux używam KDevelop'a 3 (ściślej to są chyba predefiniowane zestawy opcji czy jak to nazwać, zmodyfikowałem sobie ustawienia „KDEvelop:Skrypty"; w Notepad++ warto ustawić `Język→PHP` jeśli sam nie wykryje po rozszerzeniu pliku). Zasadniczo każdy z kolorowaniem składni i opcją zwijania kodu się nada, byle można było ustawić poniższe elementy. Jak ktoś sobie skonfiguruje inny i będzie z nim szczęśliwy – jego rzecz.

Wcięcia – tabulator zamieniany na 4 spacje. Z końca wiersza usuwamy zbędne spacje.

Notepad++: `Ustawienia→Inne`. Usuwanie spacji: `Makro→Trim trailing and save`

KDevelop: `Ustawienia→Konfiguracja edytora→Edycja` oraz `Wcięcia`.

Kodowanie – bezwzględnie UTF-8.

Notepad++: `Format - koduj w utf-8`

KDevelop: nie znalazłem, ale to chyba jakieś globalne ustawienie systemu operacyjnego jest... zawsze można też użyć `Zapisz jako`. Mieszanie UTF-8, iso-8859-2 (iso latin 2, popularnie zwane „izolatką") i windzianego cp-1250 ma opłakane skutki. Przykładowo Winblows czasem dopisuje jakieś krzaczki wyglądające jak „dtz" na początku pliku zapisanego w UTF-8. A wsadzenie tego na początek pliku php, nawet przed tag <?php sprawi że jest to traktowane jak html, wysyłane do usera. A skoro user już coś dostał, to nie ma możliwości wysłania mu ciasteczka czy informacji (http header), choćby że użyto kompresji GZIP i zobaczy ładne krzaczki.

Znak nowej linii – używamy unixowego, tzn. LF (nie CR LF!). Linuksiarze mają spokój, reszta klika w Notepad++: `Format - konwertuj na format UNIX`. Windziarzu, jeśli masz wątpliwości:

- zaznacz `Widok - Pokaż białe spacje i tabulatory/Pokaż koniec wiersza` i sprawdź czy wszędzie jest LF,
- na wszelki wypadek poproś linuksiarza o przepuszczenie plików przez programik `dos2unix`.

4.2 Narzędzie porównywania plików

W sumie to złe miejsce na to, ale się wiąże z konfigiem edytora więc co mi tam. Zastosowanie złego kodowania albo znaku nowej linii da w rezultacie brednie w stylu „zmieniono 947 linii” (czytaj: wszystkie) i zwyczajnie utrudnia życie.

Pod GNU/Linuxa zawsze i wszędzie dostępny jest `diff`, w KDE używam graficznej nakładki o nazwie `kompare`. Dla windziarzy prosty w użyciu jest `WinMerge`. `Diff/kompare` nie zrozumieją że coś nie tak jest z kodowaniem lub nową linią (można wybrać kodowanie ale oba pliki mają mieć identyczne). Konfiguracja `WinMerge`:

- `Edit - Options - Compare` – jeśli chcesz sobie włączyć ignorowanie różnic w spacjach, tabulatorach, nowej linii,
- `Edit - Options - Editor` – jeśli będziesz coś edytował na żywca z poziomu Winmerge, to ustaw tu też 4 spacje. I warto podpiąć pod „zewnętrzny edytor” Notepad++.

O porównywaniu plików trzeba skrobnąć więcej przy okazji Subversion.

4.3 Właściwa edycja

Poza wymienionymi elementami stosują się też odpowiednio zasady opisane w rozdziałach „konwencja dla pliku *.php” itd!

Nazwy – oficjalnym językiem projektu jest angielski. Nazwy zmiennych, funkcji, klas, komentarze staramy się definiować po angielsku. Do tego dochodzą nazwy uświęcone tradycją programistów – `i,j,k` dla iteratorów, `foo` czy `bar` na „robi cokolwiek” itd. Choć lepiej by `foobarów` nie było (stosujmy logiczne nazwy, bo potem ktoś będzie 5 minut myślał co tu się dzieje). Niech długość nazwy odzwierciedla zakres jej użycia (lokalne króciutkie, globalne – jeśli już muszą być – powinny być lepiej opisane).

Stosujemy garbatą konwencję nazw, tzn. `$toJestNazwaZmiennej`. Tak samo nazywamy funkcje, natomiast klasy (nie obiekty!) definiujemy z `DużejLiteryKażdyWyraz`. Najchętniej bym egzekwował pisanie memberów klas od _podkreślnika, a metod jak są, ale raz że taka przeróbka tak dużego projektu jest nierealna, dwa że byście mnie zlinczowali za pisanie wszędzie `$player → _id`. Trudno, przeżyję.

Nazwy zmiennych (z wyjątkiem takich o krótkim czasie życia jak iteratory pętli) powinny odzwierciedlać typ zmiennej:

```
$mixVar - zmienna dowolnego/modyfikowanego w trakcie pracy typu
$intVar - zmienna całkowita
$fltVar - zmienna zmiennoprzecinkowa
$blnVar - zmienna logiczna
$strVar, $chrVar - łańcuch znakowy
$arrVar - tablica znaków
$objVar - obiekt
name_class.php - plik z klasą
```

Nie stosujemy `o_takiej_podkreślnikowej_konwencji_nazywania` ani jakichś jej wariantów! Wyjątkiem na tą chwilę są pluginy Smarty'ego, ale na to wpływu nie mam (poza zmianą systemu szablonów w swoim czasie).

Operatory – jak komu wygodnie. Osobiście stawiam spacje wokół każdego `-`, `*`, `+=` itd. i uważam, że tak jest przejrzyściej (to się tyczy również strzałki `$player -> id`). Wyjątkiem jest `$i++` oraz nawiasy (nie stawiam spacji po otwierającym ani przed nawiasem zamykającym)

Struktury kontrolne (`if`, `for`, `while`, bloki w `switch`, ciała funkcji...) - 1 poziom (czyli 4 spacje) zagnieżdżenia „mięska” w stosunku do otoczenia, 1 spacja między słowem kluczowym a nawiasem okrągłym. Thindil stosuje nawiasy klamrowe zawsze, nawet gdy składnia języka tego nie wymaga (tzn. poniżej jest tylko jedna linia kodu albo inna struktura kontrolna). Szczerze

mówiąc, wszystko mi jedno (pisząc nie dla Vallheru zawsze otaczałem klamrami tylko to co było niezbędne, ale tu już przywykłem do tej konwencji). Umówmy się, że robimy jak nam wygodnie, ale spójny styl zachowujemy w całym pliku (nie ma „raz tak, raz tak"). Czyli jeśli wprowadzasz zmianę do pliku X, to albo się dostosuj, albo przerabiasz cały plik. Tylko nie róbmy jaj:

- żeby nikt się nie chwalił, „łooo, zoptymalizowałem i odchudziłem plik o 20 linijek" a potem się okazuje że to było wywalenie klamr, spacji wokół operatorów i jedno `i++`,
- i żeby nie było „wojen edycyjnych" tzn. walki na style.

Nawet nie chodzi o to, że to niepoważne. Utrudnia to czytanie kodu i porównywanie zmian.

Natomiast straszliwie, organicznie wręcz mnie wku***a konwencja z klamrą w tej samej linii:

```
if (cos == 1) {
      // bla bla bla
}
```

Nie wiem czemu tak mam, ale doprowadza mnie to do szewskiej pasji. Umówmy się, że jeśli już stosujemy klamry, to „po thindilowemu". Jak są w pionie w tej samej kolumnie, to krótkie rzeczy widać wprost co jest od czego, a dłuższe elegancko wyglądają po kliknięciu przycisku zwijania bloku tekstu.

Wywołanie funkcji – bez spacji między nazwą funkcji a nawiasem. Zmienna, przecinek, spacja, zmienna, przecinek, spacja... Znaku & (referencji) używamy wyłącznie w definicji funkcji, nigdy w wywołaniu (PHP4 łyka takie coś, PHP5 też ale wali Notice'm).

Argumenty domyślne umieszczamy na końcu – to nie jest moje widzimisię tylko wymóg składni języka (potem kompilator nie wie który argument przekazujemy, a który miałby być domyślny).

```
$var = foo($bar, $bar1);

function foo($bar, $bar1 = 0)
{
      return $bar + $bar1 + 1;
}
```

Komentarze - można stosować składnię Thindila do ważniejszych rzeczy, jak opisy działania funkcji czy główne if'y decydujące na bazie $_GET, którą podstronę gracz ogląda. Krótkie pierdółki związane z algorytmem, tłumaczące zawiłego if'a itd. można zwykłym jednolinijkowym komentarzem.

```
/**
* Example comment.
* Yet another line.
*/

// Short comment.
```

Komentarz musi być złożony ze zdań (duże litera na początku, kropki). Powinien być zwięzły i mieścić się w 1 linii na normalnym ekranie bez zawijania (co w dobie rozdzielczości minimum 1024x768 i przy kodzie wyświetlanym zwykle czcionką 10 oznacza ok. 80 znaków), zawsze można go pociąć na kilka zdań/linii. Ustaw sobie „linijkę" na mniej więcej tej odległości. Ten wymóg ulegnie zmianie gdy poznam PHPDOCUMENTOR i się zdecyduję na stosowanie jego lub Doxygen'a. Póki co staraj się poświęcić 1 linię komentarza na opis znaczenia każdego argumentu wejściowego funkcji, jeśli definiujesz jakąś nową. A przynajmniej choć krótko ją omówić.

4.3.1 Konwencja dla *.php

Zawsze stosujemy pełne znaczniki kodu, tzn `<?php ... ?>` a nie `<? ... ?>`. Przy skopanej konfiguracji Apache stosowanie drugiego stylu rodzi konflikty gdy serwer zobaczy początek tagu XML (XHTML itd) `<?xml`...

```
<?php
/**
 *   File functions:
 *   Class to get data about outposts, fight, collect taxes, equip veterans etc.
 *
 *   @name                 : outpost_class.php
 *   @copyright            : (C) 2007 Orodlin Team based on Vallheru Engine 1.3
 *   @author               : eyescream <tduda@users.sourceforge.net>
 *   @version              : 0.1
 *   @since                : 15.06.2007
 *
 */

//
//
//       This program is free software; you can redistribute it and/or modify
//   it under the terms of the GNU General Public License as published by
//   the Free Software Foundation; either version 2 of the License, or
//   (at your option) any later version.
//
//   This program is distributed in the hope that it will be useful,
//   but WITHOUT ANY WARRANTY; without even the implied warranty of
//   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//   GNU General Public License for more details.
//
//   You should have received a copy of the GNU General Public License
//   along with this program; if not, write to the Free Software
//   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
//
// $Id$

// All database SELECT's assume that we're working in numeric fetch mode, for example ADODB_FETCH_NUMERIC is set.
/// TODO: migration to php5 (public/private)
class Outpost
{
    var $id;
```

Rysunek 1: Nagłówek pliku PHP.

Po „File functions” umieszczamy krótki opis za co plik jest odpowiedzialny. Następnie mamy komentarze zgodne z notacją Doxygen'a. Na razie są takie, może się niedługo zmienią. Znaczenie jest dość oczywiste, wpisujemy prawdziwą nazwę pliku, zmieniamy rok koło „copyright” jeśli trzeba. Skoro nastąpiło scalenie z Vallheru i gra się nazywa Orodlin ale silnik to dalej Vallheru, to trzeba się zastanowić nad nazwą...

Linijek @author może być więcej niż jedna, siebie dopisujemy na końcu pamiętając o adresie e-mail. Bieżąca wersja silnika, data oddania do użytku (wrzucenia na serwer, subversion itd).

Dalej mamy informację o licencji. Zastanawiam się czy tego nie wywalić i nie zastąpić np. „Published under GNU GPL 2 or later. See /install/COPYING file for license details.” Trochę nadmiarowo toto w każdym pliku. Dalej mamy linijkę ze „znacznikiem” `$Id$`. Pamiętaj by w wyedytowanym pliku ta linia zawsze miała taką postać. W chwili wrzucenia na Subversion

zostaje ona zmodyfikowana, ale ta modyfikacja często się przydaje później do śledzenia zmian. Ten „wyraz” ma dokładnie tak wyglądać, bez dodatkowych spacji.

Jeśli są jakieś robocze informacje, to oznaczamy je komentarzem zawierającym słowo „TODO”. Jak widać KDevelop je elegancko podświetla jesli użyto trzech *backslash*'y albo komentarza /** tekst... */, to też jest związane z Doxygen'em (przy generacji dokumentacji na podstawie kodu on potrafi wyrzucić „znalazłem TODO w linii X pliku Y” jako ostrzeżenia, przydatna rzecz). Gdy plik idzie na serwer testowy albo do konsultacji z kimś, to TODO sobie mogą być a wręcz pomagają się zorientować co jest prowizorką. Wersja stabilna „do gry” nie powinna ich zawierać (co nie znaczy że masz wyczyścić komentarze a prowizorki zostawić :P).

Potem już normalnie, jakieś require_once, „mięsko” pliku itd. Konkrety możesz znaleźć w rozdziale opisującym co jest co.

4.3.2 Konwencja dla *.tpl

Poza pilnowaniem wcięć podobnie jak w *.php niewiele można tu napisać. Pamiętamy o umieszczeniu tagu `{strip}` tnącego zbędne spacje w wynikowym html:

- jeśli plik zawiera deklarację `<!DOCTYPE>` dla XHTML, to `{strip}` umieszczamy zaraz za nią,
- w przeciwnym przypadku `{strip}` jest w pierwszej linii,
- a na końcu pliku stawiamy `{/strip}`.

Podobnie jak dla *.php, trudniejsze elementy warto opatrzyć komentarzem:

```
{* O właśnie takim *}
{* albo
      {wielo}
linij
kowym*}
```

Z tej sztuczki można korzystać zakomentowując kod który sprawia problemy. Można otoczyć komentarzem inne tagi Smarty'ego, np. pozbywając się `{strip}` gdy coś nam się kaszani w wynikowym html i chcemy go dokładniej obejrzeć. Oczywiście na serwer gry plik powinien trafić ze strip'ami i bez zakomentowanych prowizorek.

Konwencja dla *.css i *.js

Nie mam pojęcia co ustali Aranwe to się wymądrzał nie będę. To samo z Javascript – dopiero się uczę, za pół roku będę wiedział jakie kretynizmy popełniam obecnie. Na razie robimy „byle działało”.

5 Optymalizacja i efektywność

W tym rozdziale ciężko jest czytać pojedynczo artykuły. To się wszystko wiąże ze sobą, poszczególne sztuczki wpływają na siebie i finalnie pozwalają zastąpić stronę kodu trzema linijkami. Z drugiej strony – **nie próbuj tego czytać na jedno posiedzenie**, bo mętlik straszliwy wyjdzie.

Zresztą w ogóle proces optymalizacji ma charakter iteracyjny – odchudzi się plik o 1/3 to dopiero wtedy widać, co jeszcze się da wycisnąć. Wstawiłem w marcu licznik ile poziomów Strażnicy można aktualnie kupić i dumny jak paw łaziłem, że już się nie naciska durnego F5, transferu nie marnuje tylko wybiera z listy. Minęły 3 miesiące i z powrotem siedzę nad tym kodem, przerabiam, kombinuję...

Dołożyć „zamiast zakonczenia” o szanowaniu czasu kodera i serwera.

Tekst bazuje na własnych doświadczeniach oraz:

- http://www.strefaphp.net/artykul46.html
- http://webcity.pl/webcity/artykuly.php/t/20

5.1 Ogólne sztuczki

Poniższy tekst to żadna filozofia, ale warto zebrać ogólnie znane dobre rady w jednym miejscu, na przykład żeby potem przejrzeć swój kod na spokojnie przed wrzuceniem na serwer.

- zamiana `$i = $i + 1` na `$i++` a `X = X + Y` na `X += Y`. I analogicznie: `-`, `*`, `/`, `%`, operatory logiczne...

- Jeśli coś ma być liczbą, to niech będzie. Nie mnożyć przez „`7`" ani `'7'` tylko przez `7`, bo PHP musi wcześniej się bawić w jakieś rzutowanie typów (to się tyczy również porównań, także na poziomie MySQL – jeśli coś nie jest stringiem to ani nie insert'ować ani nie select'ować z użyciem apostrofów).

- `max()`, `min()` i operator warunkowy zamiast mnóstwa kodu:

```
$x = $y + 10 * $z;
if ($x > 100)
{
      $x = 100;
}
// równoważne:
$x = min($y + 10 * $z, 100);  // można wiele argumentów: min($a,$b,8,...)

if ($a > 100)
{
      $x = coś_gdy_prawda;
}
else
{
      $x = coś_innego;
}
// równoważne (nawet nawias można usunąć i dalej działa):
$x = ($a > 100) ? coś_gdy_prawda : coś_innego;
```

- jeśli już musisz w funkcji używać zmiennych globalnych (przykład ekstremalny: kowal.php, funkcja `createitem()`), to przynajmniej oszczędź miejsce wypisując je w jednej linii:

```
   global $db;
   global $player;
   global $arrItem;
   global $intAbility;  // i tak 12 linii
   ...
   global $db, $player, $arrItem, $intAbility, $intItems, ...;
```

- W if'ach, warunkach końca pętli itd. elementach grupuj mądrze. Jeśli warunek zawiera &&, to „częściej fałszywą" część umieść przed &&. Jak będzie sprawdzał, to stwierdzi że jest fałszywe i nie będzie sprawdzał drugiego członu, bo i tak „AND" z zerem to zero. Odwrotnie

z || („OR”) - jak pierwszy element będzie prawdą to wiadomo że drugi można olać i PHP faktycznie olewa. To są niby drobiazgi, ale podobnie powinno się pisać WHERE do zapytań. Tzn. tak dobierać kolejność by pierwsze stwierdzenie po WHERE jak najmocniej wycięło zbiór wyników, żeby następne testy nie leciały po całej tabeli czy jej 10%, tylko np. pięciu wierszach.

– Tekst o tej „częściej fałszywej” części dotyczy też wyższości porównywania <,> nad >=, <= - po co nam 2 porównania jak można mieć jedno? (Choć szczerze mówiąc było to aktualne jakieś 15 lat temu... Intel się zorientował że programiści nie umieją pisać porównań to się zlitował i od 386 poszerzył listę rozkazów procesora. Ale nie zaszkodzi ;))

– Jeśli masz `switch` i program zwykle ląduje w sekcji `default` (umieszczonej na szarym końcu), to rozważ zdefiniowanie ich *explicite* i przeniesienie najczęściej trafianych elementów na przód `switch`'a. A jak jest krótki (powiedzmy, 2 warunki + default) to może `if` się lepiej sprawdzi?

– nic na siłę. Jeśli kopiujesz jakiś fragment kodu, zastanów się nad wydzieleniem go do jakiejś funkcji i wywoływaniem go. Za czas jakiś trzeba będzie coś przerobić i się zacznie panika i latanie po dziesięciu plikach. Technika CTRL+ Copy/Paste/Find/Replace doprowadziła Vall do przytykania się przy 70 graczach.

– Przeczytaj jeszcze raz poprzedni punkt☺ Strasznie trudne jest odwrócenie takiego procesu kopiowania i zamieniania drobiazgów. Obecnie tworzenie komponentów astralnych jest pchnięte do includes/artisan.php, ale kosztowało mnie to 3 dni siedzenia nad wszystkimi plikami „rzemieślniczymi”. W międzyczasie Dellas dodawał astralne jubilerstwo do banku, złodzieja, klanu, rynku itd. metodą „kopiuj-wklej”. Gdy już tylko brakowało faktycznego wytwórstwa przez jubilera, musiałem jedynie zdefiniować tabelkę zużycia surowców zgodną z projektem i wstawić wywołanie odpowiedniej funkcji. Poezja.

– Jeśli budujesz jakąś skomplikowaną konstrukcję, zastanów się co chcesz osiągnąć. Naszego kodu powinno być tak naprawdę jak najmniej. Sumujesz elementy tablicy to nie leć pętlą tylko użyj `array_sum()` (albo przetwórz kolumny w zapytaniu SQL). Ideałem jest gdy nasz kod składa się niemal wyłącznie z jakichś wywołań systemowych czy funkcji wbudowanych w język – wtedy jest to i szybsze niż cokolwiek, co byśmy wymyślili, i łatwiejsze w debugowaniu. Ta uwaga odnosi się również do wielu funkcji przetwarzania tekstów – działanie funkcjami wbudowanymi w język będzie szybsze niż cokolwiek co napiszemy ręcznie. Postaram się wypisać kilka użytecznych przykładów, ale nic nie zastąpi Google i manuala PHP.

Ciężko to osiągnąć, ale walczyć trzeba – *vide* ranking.php (tak naprawdę plik się składa z jednego wypasionego zapytania i trochę Smarty'ego. A zaczynaliśmy od dodatkowej tabeli w `players` i aktualizacji co reset).

– Szanuj bazę danych i transfer. Dużo zapytań da się opóźnić, scalić itd. połączyć rzeczy które dotyczą jednej tabelki i jednego gracza. Thindil w wielu miejscach dość rozrzutnie pchał zapytania wykonując np. 3 osobne UPDATE do tej samej tabeli i z tym samym WHERE – bezsens. To samo się tyczy tego co wyświetlamy – nigdy nie wkurzała Cię konieczność durnego klikania „odśwież" w ekwipunku, sklepie w strażnicy itd? Wystarczy chwilkę poświęcić i poprzenosić parę linijek...

– Teksty zapisujemy w 'pojedynczych apostrofach', nie w „podwójnych". I takie w plikach językowych, i takie w pliku głównym. Wyjątkiem jest kilka funkcji, które się zabawnie zachowują po podaniu 'takiego parametru': `implode()`, `explode()`, `wordwrap()` (może są i inne). Kolejny wyjątek to atrybuty html, np `<form method="post"...>` - tu zawsze cudzysłów.

Konkatenacja łańcuchów tekstów (tzn. łączenie kropkami zamiast również działających znaków dodawania). Nazwy tabel i kolumn bazy danych zapisujemy w odwróconych apostrofach (ang. *backtick*, po polsku „ciapki") - o `tych`, razem z tyldą ~ pod klawiszem Esc.

– Unikać `echo()`. Jeśli już musisz ale to kurna musisz wyrzucić coś na ekran i to musi iść przez `echo()` a nie przez templejty, to użyj `print`. `Echo` ma to do siebie, że wymusza wysłanie na ekran/do użytkownika. Tzn. te dane będą lecieć przez dysk, kartę sieciową itd. zaraz po trafieniu na tą instrukcję. Zawalamy sobie sieć marnując pakiety na takie pierdzenie co chwilka jakimś krótkim tekstem, marnujemy czas procesora na wysyłkę zamiast „raz a dobrze". Wprawdzie my mamy włączone buforowanie wyjścia, które „pochłania" takie echa, ale ono działa tylko jeśli włączyła się kompresja gzip. A jak ktoś ma starą przeglądarkę, nie obsługującą tego, to po co mamy sobie tak utrudniać... Jak musi być, to `print`. (oczywiście to dotyczy kodu lądującego na serwerze gry. U siebie do debugowania można robić co się chce).

Pier***ony Smarty używa co chwilę `echo()` w tych swoich skompilowanych plikach z katalogu templates_c... Trzeba go kiedyś przekonfigurować a jak się nie da to wyje**ć.

– Słuchać starych mądrości informatyków, takich jak: „90% czasu działania program spędza w 10% swojego kodu", „kod nie jest dobry zanim nie będzie przepisany 3 razy", „jak nad czymś siedzisz i nie widzisz rozwiązania, to spytaj kolegi", metodologia KISS: „keep it simple and stupid"...

5.2 Pętle

Znamy 3 typy pętli: `while`, `do...while` i `for` (foreach jest opisana w rozdziale poświęconym tablicom). Co to jest pętla – odsyłam do Google i jakiegoś kursu PHP. `While` i `for` to w zasadzie to samo – warunek końca pętli jest testowany **przed** rozpoczęciem, więc może się zdarzyć że do pętli w ogóle nie wejdziemy! `For` się różni tylko tym, że ma „miejsce" na inicjację i „wyrażenie po każdym obrocie pętli" - oszczędność linijek kodu, ale to kosmetyka. `Do...while` wykona swoje ciało co najmniej raz i dopiero potem będzie testować warunek. Tyle tytułem przypomnienia.

```
Przykład z outposts.php – naliczanie zysku z danin w strażnicy. 17 linijek.
$intGaingold = 0;
for ($i = 0; $i < $_POST['amount']; $i++)
{
      $intGain = rand(1,5);
      $intGaingold = $intGaingold + ($intArmy * $intGain);
}
$fltBonus = ($intGaingold * ($out -> fields['btax'] / 100));
$intBonus = round($fltBonus, "0");
$intGaingold = (int)($intGaingold + $intBonus);
$intFatigue = $out -> fields['fatigue'] + (10 * $_POST['amount']);
if ($intFatigue > 100)
{
      $intFatigue = 100;
}
$db -> Execute("UPDATE outposts SET fatigue=".$intFatigue." WHERE id=".$out ->
fields['id']);
$db -> Execute("UPDATE outposts SET gold=gold+".$intGaingold." WHERE id=".$out
-> fields['id']);
$db -> Execute("UPDATE outposts SET turns=turns-".$_POST['amount']." WHERE
id=".$out -> fields['id']);
```

Pętle mają tą niemiłą własność, że trzeba je pisać starannie. Każda mikrosekundowa niestaranność przejechana w pętli 1000 razy zaczyna być odczuwalna. Poza tym – gdy kod chudnie i więcej się go mieści na ekranie, to łatwiej błędy wyłapywać. No to lecimy.

1. Elementy początkowe pętli `for(...;...;...)`, podobnie jak to co na końcu, można łączyć przecinkami. Czyli najzupełniej poprawne jest zgrupowanie warunków początkowych w postaci `for ($i = 0, $intGaingold = 0; $i < $_POST['amount']; $i++)`. Pierdółka, ale zawsze to 1 linia mniej, Eye się cieszy :P
2. Warunek, czyli $i < ilości zbieranych podatków – ma być tak szybki i prosty jak to tylko możliwe. Prosty, żeby takie cudeńka wewnątrz procesora Intel Pentium 2 i lepszych mogły z tym warunkiem coś mądrego zrobić. Ja o przetwarzaniu potokowym, predykcji skoków itd. mogę bredzić długo i zawile. Ważne by zapamiętać że ma być proste. Tu by nie

zaszkodziło wsadzenie zwykłej zmiennej zamiast pobrania z tablicy `$_POST` (skopiowanie do jakiejś), ponieważ pobranie tablicowe trochę więcej kosztuje. Ale to już takie czepiactwo maksymalne, bywają gorsze sytuacje (będzie później).

3. `$i++` to taka kolejna pierdółka. Dla `$i` równego zero `echo $i++` wyświetli zero, a **po** tym poleceniu `$i` będzie miało wartość 1. ++ jest za zmienną, czyli oznacza „zwróć jako wynik **starą** wartość i się powiększ »po użyciu«". To jest taka mała głupota. Zastanówmy się jak by wyglądała taka funkcja robiąca to samo co `$zmienna++`:

```
function plusplus(&$i)  // tym & się nie przejmować (referencja)
{
      $stareI = $i;
      $i = $i + 1;
      return $stareI;
}
```

Na co to komu? To mogłaby spokojnie być jedna linijka: `return $i = $i + 1;`. Do niczego ta stara wartość nam nie potrzebna, nawet na „echo" nie idzie – jest tylko jako wyrażenie po każdym obrocie pętli. Więc warto zamiast tego pisać `++$i` (tak, taka dziwna składnia działa i to zgodnie z oczekiwaniami :P).

4. Czas na „mięsko" pętli. Każde przypisanie zmiennej kosztuje, a tu widać kilka spraw:
   - `$intGain` jest zbędna – można do drugiej linijki wstawić bezpośrednio `rand(1,5)` i zadziała.
   - X = X + Y jest wolniejsze od konstrukcji X += Y. To jest coś podobnego do wyższości $i++ (czy ++$i) nad $i = $i + 1. Przy okazji wywalimy nawias wokół mnożenia – jest przecież kolejność wykonywania działań, a takie nawiasowanie to dodatkowa robota dla kompa.
   - dostajemy ostatecznie w pętli `$intGaingold += $intArmy * rand(1,5);`
5. Teraz taka delikatna sprawa jak „niezmiennik pętli" (ang. *loop invariant*). Zasadniczo chodzi o wypychanie rzeczy, które nie zmieniają się w trakcie działania pętli, gdzieś poza nią. Jeśli da się coś zrobić raz zamiast N razy, no to czysty zysk. Niezmienniki dość łatwo się wyłapuje, ale jednak trochę praktyki to wymaga (czasem trzeba zmienić sposób myślenia, przebudować algorytm itd). Tu takim niezmiennikiem jest mnożenie przez ilość armii wysłanej do wiosek – będzie jakaś tragedia jak ten zysk `$intGainGold` przemnożę przez ilość wojska `$intArmy` raz, po wyjściu z pętli? Przecież wyjdzie na to samo.

   Kandydaci na niezmienniki pętli to wszelkie stałe. Wszelkie rzeczy, które da się wyliczyć raz. Np. Dellas wymyślił, że ta ilość kasy z podatków ma być na randomie, ale randomie

zależnym od rozmiaru Strażnicy. No to pchnąłem to obliczenie argumentu funkcji `rand()` przed pętlę, czego tu nie widać dla prostoty przykładu. Potem mam tylko wywołany `rand(2, $intModifier)` i wszyscy są zadowoleni. Ogólnie – trochę praktyki i jest ok.

6. Po zastosowaniu wskazówek ogólnych z początku rozdziału (i przeróbce na klasę, o czym w swoim czasie) dostajemy 8 linii:

```
for ($i = 0, $intGaingold = 0; $i < $intTimes; ++$i)
{
      $intGaingold += rand(1,5);
}
$intGaingold *= ($this -> warriors + $this -> archers) * (1 + $this -> bonusTax
/ 100);
$this -> gold += round($intGaingold);
$this -> fatigue = min($this -> fatigue + 10 * $intTimes, 100);
$this -> db -> Execute('UPDATE `outposts` SET `fatigue`='.$this -> fatigue.',
`gold`='.$this -> gold.', `turns`='.($this -> turns -= $intTimes).'  WHERE
`id`='.$this -> id);
```

Możnaby to zwinąć do jeszcze mniejszej ilości linijek, np. zostawiając tylko pętlę (z pustym ciałem, tzn. taką która ma tylko kolce `{}` a „`+=`” jest wpisane po przecinku tam, gdzie jest `++$i`) i długie wyrażenie do bazy danych najeżone nawiasami. Ale to już by był dość nieczytelny kod, nie jest źle tak jak jest. Natomiast czy by się i pętli nie pozbyć biorąc jakąś wartość średnią... to już nie powinna to być decyzja kodera tylko projektanta ☺

7. Jeszcze jeden dość ważny przykład się wiąże z pętlami i niezmiennikami. Standardowo na szybko pętlę po tablicy pisze się tak:

```
for ($j=0; $j<sizeof($arr); $j++)
      echo $arr[$j]."<br>";
```

Rozmiar tablicy (mierzony `count()` lub `sizeof()`) jest niezmiennikiem i nie ma sensu zliczać ilości elementów co chwilę, w kazdym badaniu warunku końca pętli. Do tego wyświetlenie (zapis do bazy, cokolwiek innego) można opóźnić:

```
for ($j=0, $max = sizeof($arr), $s = ''; $j<$max; $j++)
      $s .= $arr[$j].'<br>';
echo $s;
```

(ten przykład jest o tyle kiepski że możnaby w ogóle tą pętlę wywalić i zastosować

```
echo implode(„<br>„, $arr).'<br>';
```

No ale to już specyficzność danego zastosowania. Zwykle nie chodzi o `echo` i wyświetlanie tylko o jakieś działania w pętli. A od wyświetlania to jest Smarty :P).

8. Jeśli wynik działania w pętli nie zależy od kolejności wykonywania (np. przemnożyć każdy element tablicy przez 2), to rozważ puszczenie pętli w dół, tzn. `for($i=max; $i; $i--)`. Zwłaszcza środek wygląda dziwnie, ale to znaczy że zatrzyma się na zerze (zero jest rzutowane do `false` i przestaje spełniać warunek wyjścia z pętli, wszystkie inne wartości są jak `true`), czyli tak jak trzeba ;)

5.3 Efektywne ADOdb

Tekst dotyczył będzie też ADOdb Lite (jeśli się zdecydujemy na zamianę), jak również innych podobnych systemów (PDO dostarczone z PHP5) czy nawet MySQL pisanego „na żywca" z PHP bez warstwy pośredniej. Chodzi o pewne konwencje, sposoby myślenia itd. Jeśli jakaś funkcja będzie miała inną nazwę – nieważne.

Zacząć należy od linka do dokumentacji ADOdb. Po rozdziale o pętlach wiemy już, że są one złe i należy je optymalizować ile wlezie. Zobaczymy co się da wycisnąć z ADOdb.

1. Pobranie danych do tablic. W PHP wyróżniamy tablice w których indeksami są liczby i łańcuchy tekstu (tzw. tablica asocjacyjna). Różnica między `$tab[0]` a `$tab['gruszka']` jest dość oczywista ;) Wygoda PHP polega na tym że możemy toto mieszać nawet w obrębie jednej tablicy. Ale niestety szybszy dostęp mamy do tablic indeksowanych tradycyjnie. Więc pierwsze co robimy przy tworzeniu nowego pliku/całkowitej przeróbce istniejącego to zmieniamy tryb pobierania danych z bazy z asocjacyjnego na numeryczny:

```
$ADODB_FETCH_MODE = ADODB_FETCH_ASSOC;
$rs1 = $db->Execute('SELECT `kol1`, `kol2` FROM `tabela`');
      // $rs1->fields['kol1'] zawiera 7, $rs1->fields['kol2'] zawiera 13
$ADODB_FETCH_MODE = ADODB_FETCH_NUM;
$rs2 = $db->Execute('SELECT `kol1`, `kol2` FROM `tabela`');
      // $rs2->fields[0] zawiera 7, $rs2->fields[1] zawiera 13
```

Różnica jest zasadnicza. To **musí** być zmieniane z głową, bo wszędzie w kodzie Thindila gdzie mamy `fields[coś]` to „coś" jest asocjacyjne i nie zaskoczy jak się zmieni na numerki. Tylko całe pliki zmieniać, albo co trochę i przełączać, posuwając się jakby spójnymi fragmentami.

Jest toto szybsze, natomiast wymaga staranniejszego pisania kodu. Jeśli mamy SELECT ośmiu kolumn to dodanie dziewiątej nie jest trudne, pod warunkiem, że się ją w zapytaniu dopisze na koniec. Gorzej gdy trzebaby ją kiedyś wstawić do środka, żeby ładnie i logicznie szły po kolei. Zmiana indeksów w algorytmie długim na 2 ekrany → przerąbane. Wtedy jedyny ratunek to porządnie nazwane zmienne, dobre komentarze kodu itd.

2. Po drugie trzeba się zastanowić czy cały ten obiekt zwracany jako rezultat `$db → Execute()` jest nam potrzebny. W dokumentacji nazywają go RecordSet'em, ma tam parę fajnych rzeczy: ilu wierszy tabeli dotyczyło zapytanie, ma funkcję `MoveNext()` no i najczęściej używane pole - „`fields`". A jak by tak olać te zbędne dodatki i otrzymywać z bazy danych tylko 1 wiersz albo i całą tabelkę dwuwymiarową? Nawet bez tej ilości

wierszy można żyć, `count()`'a się raz machnie i spadówa na drzewo. Najpierw prostszy przykładzik – pobranie jednego wiersza z tabelki players:

```
$arrInfo = $db -> GetRow('SELECT `id`, `user`, `level`, `credits` FROM `players`
WHERE `id`=1');
if (!empty($arrInfo))   // jeśli tablica nie jest pusta
{
      print 'ID: '.$arrInfo[0].'<br />';
      print 'Złoto w sakiewce: '.$arrInfo[3].'<br />';
}
```

Prawda, że wygodniej niż się babrać z masą „`$objQuery → fields`”? `GetRow` pobiera jeden wiersz, zawartość kolumn jest dostępna wprost w zwracanej tablicy. Jeśli *fetch mode* jest ustawiony na asocjacyjny, to byśmy pisali `$arrInfo['id']` itd. Tutaj założyłem, że jest `ADODB_FETCH_NUM`, więc dane idą w kolejności takiej jaka jest w SELECT'cie (w asocjacyjnym by to nie miało znaczenia).

Drugi przykład – pobieramy prawdziwą macierz, wiersze to poszczególni gracze, kolumny to ich dane. Powiedzmy że pobraliśmy całość, a chcemy wyświetlić poziom siódmego gracza (zakładamy że to siódmy wiersz, że on istnieje, że nie urządza nas wstawienie do SELECT odpowiedniego WHERE itd).

Przykład beznadziejny:

```
$ADODB_FETCH_MODE = ADODB_FETCH_ASSOC;
$objQuery = $db -> Execute('SELECT `id`, `user`, `level` FROM `players`');
$objQuery -> MoveNext();
$objQuery -> MoveNext();
$objQuery -> MoveNext();
$objQuery -> MoveNext();
$objQuery -> MoveNext();
$objQuery -> MoveNext();
print $objQuery -> fields['level'];
```

Przykład mało zły:

```
$ADODB_FETCH_MODE = ADODB_FETCH_ASSOC;
$objQuery = $db -> Execute('SELECT `id`, `user`, `level` FROM `players`');
$objQuery -> Move(6);
print $objQuery -> fields['level'];
```

Przykład przyzwoity:

```
$ADODB_FETCH_MODE = ADODB_FETCH_ASSOC;
$arrResult = $db -> GetAll('SELECT `id`, `user`, `level` FROM `players`');
print $arrResult[6]['level'];
```

Szprytne!

```
$ADODB_FETCH_MODE = ADODB_FETCH_NUM;
$arrResult = $db -> GetAll('SELECT `id`, `user`, `level` FROM `players`');
print $arrResult[6][2];
```

3. Żeby była jasność: *fetch mode* i użycie `GetRow/GetAll` itd. zamiast `Execute` to są dwie niezależne rzeczy. Po prostu najefektywniej jest je stosować razem. Tryb można ustawiać albo zmieniając wartość tej zmiennej z przykładów, albo używając funkcji `SetFetchMode()`. Funkcja zwraca poprzednio używany tryb:

```
$oldFetchMode = $db -> SetFetchMode(ADODB_FETCH_NUM);
// Nasze zapytania na fetch'u numerycznym...
// ...
// ... i powrót do starego trybu, jeśli trzeba.
$db -> SetFetchMode($oldFetchMode);
```

4. `GetRow/GetAll` i podobne świetnie zastępują `Execute` tylko przy zapytaniach SELECT. To oczywiste ale *pro forma*: nie próbujcie tego używać do INSERT, UPDATE, DELETE ☺

Do tych zapytań można stosować `_Execute` albo `_query`, pod warunkiem że mamy doinstalowane rozszerzenie do PHP wspomagające ADOdb. Nie wiem czy zostaniemy przy ADOdb, a nawet jeśli zostaniemy to rozsianie takich `_query()` po silniku Vallheru sprawi, że będzie on wymagał przerabiania w celu użycia na zwykłych hostingowych serwerach. Czyli dla wielu będzie bezużyteczny (na hostingach nie można grzebać przy konfigu PHP).

5. Przykład już z życia wziętej optymalizacji. Rynek przedmiotów (imarket.php), dodawanie oferty, 21 linii:

```
$rzecz = $db -> Execute("SELECT `id`, `name`, `amount`, `power`, `zr`, `szyb`
FROM `equipment` WHERE `status`='U' AND `type`!='I' AND `owner`=".$player ->
id);
$arrname = array();
$arrid = array(0);
$arramount = array();
$i = 0;
while (!$rzecz -> EOF)
{
      $arrname[$i] = $rzecz -> fields['name'];
      $arrid[$i] = $rzecz -> fields['id'];
      $arramount[$i] = $rzecz -> fields['amount'];
      $arrPower[$i] = $rzecz -> fields['power'];
      $arrAgi[$i] = $rzecz -> fields['zr'] * -1;
      $arrSpeed[$i] = $rzecz -> fields['szyb'];
      $rzecz -> MoveNext();
      $i = $i + 1;
}
$rzecz -> Close();
if (!$arrid[0])
{
    error(NO_ITEMS);
}// Dalej mają miejsce $smarty -> assign tabelek i komunikatów
```

Do kompletu trzeba pokazać kawałek kodu z *.tpl:

```
<select name="przedmiot">
{section name=item1 loop=$Name}
      <option value="{$Itemid[item1]}">
      {$Name[item1]} ({$Iamount}:{$Amount[item1]})
      ({if $Itempower[item1] != 0} +{$Itempower[item1]} {/if}
      {if $Itemagi[item1] != 0} {$Iagi} {$Itemagi[item1]} {/if}
      {if $Itemspeed[item1] != 0} {$Ispe} +{$Itemspeed[item1]} {/if})
      </option>
{/section}
</select>
```

Zapytanie dość biednie napisane, bo najbardziej tnące zbiór wynikowy byłoby umieszczenie za `WHERE` sekwencji „`owner='.$player->id.' AND ...`”. Dalej jest tylko gorzej – w pętli mamy jakieś durne kopiowanie wyników zapytania do tablic (no, zręczność jest zamieniana na ujemną, tzn. karę do zręczności). Nawet `$i++` nie ma. Pierwsza przymiarka (z `ADODB_FETCH_NUM`):

```
$arrItems = $db -> GetAll('SELECT `id`, `name`, `amount`, `power`, `zr`, `szyb`
FROM `equipment` WHERE `owner`='.$player -> id.' AND `status`=\'U\' AND
`type`!=\'I\'');
if (empty($arrItems))
{
    error(NO_ITEMS);
}
for ($i =0, $intMax = count($arrItems); $i < $intMax; ++$i)
{
      $arrItems[$i][4] *= -1; // przemnóż zręczność
}
```

Zaraz to do Smarty przypiszemy... ale chwila moment, w sumie to po co nam ta pętla? Nie prościej już pobrać z bazy danych zamienionej wartości?

```
$arrItems = $db -> GetAll('SELECT `id`, `name`, `amount`, `power`, -`zr`, `szyb`
FROM `equipment` WHERE `owner`='.$player -> id.' AND `status`=\'U\' AND
`type`!=\'I\'');
```

Tym sposobem całkowicie **wycinamy pętlę** i pozbywamy się masakrycznego kopiowania. Zwinęliśmy kod do 1 linii zapytania i `if`'a czy tablica nie jest pusta. Trudno by było mocniej odchudzić ten kod. Do Smarty przypiszemy sobie całą macierz:

```
$smarty -> assign('Arritems', $arrItems); //zaawansowane: assign_by_ref, str. 35
```

A kod w templejcie będzie wyglądał bardzo podobnie do istniejącego:

```
<select name="przedmiot">
{section name=i loop=$Arritems}
      <option value="{$Arritems[i][0]}">
      {$Arritems[i][1]} ({$Iamount}:{$Arritems[i][2]})
      ({if $Arritems[i][3]} +{$Arritems[i][3]}{/if}
      {if $Arritems[i][4]} {$Iagi} {$Arritems[i][4]} {/if}
      {if $Arritems[i][5]} {$Ispe} +{$Arritems[i][5]} {/if})
      </option>
{/section}
</select>
```

Proste?

6. Dalsza optymalizacja jest jak najbardziej możliwa, ale już to co mamy pięknie przyspiesza działanie i zmniejsza użycie RAM-u ;) Ten przykład z rynkiem nie jest umieszczony w grze – czeka na zmiłowanie i ogólne „uwspólnienie” kodu rynków. Ale takich miejsc z bezsensownym kopiowaniem jest w grze masa.

Dużo można uzyskać samym zapytaniem SQL, ale to materiał na osobny rozdział. Tak samo trzeba rozdybać i przygotować tutorial z cache ADOdb: komendy CacheExecute, CacheGetAll itd.

5.4 Brzytwa Ockhama

„Nowa sześciotomowa encyklopedia PWN", hasło „Ekonomii zasada":

1) *filoz. zasada prostoty, oszczędności – respektowana przez materię nieożywioną i ożywioną; w prakseologii – obowiązująca działanie ludzkie oszczędność wydatkowanych zasobów (wysiłku, energii, materiałów itd.) lub dążenie do zwiększania wydajności przy tym samym ich wydatkowaniu; w ekonomii myślenia – postulat zw. brzytwą Ockhama głoszący, że „**bytów nie należy mnożyć więcej niż potrzeba**", co powinno się wyrażać eliminowaniem zbędnych słów, pojęć, aksjomatów;[...] wymóg wyjaśniania faktów przez naukę w sposób najprostszy i postulat najprostszego formułowania opisu, twierdzeń, praw;*

2) *ekon. zasada gospodarowania – wymóg stosowania optymalnego stosunku między skwantyfikowanymi nakładami a efektami; na z.e. składają się zasada oszczędności i zasada wydajności.*

Po pierwsze, nic nie zastąpi wzięcia kartki, ołówka i zastanowienia się co chcemy osiągnąć. Nie mówię o rysowaniu diagramów UML czy cholera wie czym (chociaż przy projektowaniu nowej klasy nie zaszkodzi). **Prosta optymalizacja** jakiegoś wzorku czasem może zdziałać cuda:

- suma wyrazów ciągu arytmetycznego/geometrycznego (a jak się nie da wyznaczyć reguły rządzącej ciągiem i mamy elementy wpisane w tablicę to użyć `array_sum()` zamiast pętli). Prawdopodobieństwo sukcesu w wielokrotnej czynności sprawdzane schematem Bernoulliego – jeśli tylko się da, to lepiej mieć jedno potęgowanie i spokój niż pętlę
- zorientowanie się, że w kilku kolejnych wzorach jakiś element (mianownik ułamka, mnożenie przez coś itd.) się powtarza – wyliczenie wartości raz i potem tylko podstawianie. Im bardziej bolesne dla kompa jest to działanie (dzielenie, pierwiastek, logarytm...), tym czujniejszy powinien być programista.
- scalanie wspólnych wyrazów, redukcja nawiasów, grupowanie wyrażeń czy czego tam w podstawówce uczą – ty zmarnujesz minutę na kartką, ale serwer odrobinę odetchnie (*„zwłaszcza jak będzie to w pętli robił", że tak sobie posmęcę aż do znudzenia*), a za pół roku ktoś będzie ten kod czytał to się szybciej połapie.

Po drugie – cięcie zbędnych zmiennych i przypisań:

- trochę było widać w rozdziale 5.2 o pętlach – te wszystkie pośrednie zmienne z bonusami do danin w Strażnicy – to z randoma, to z armii, to z punktów wydanych z umiejętności dowodzenie... na cholerę tam tego tyle? Ctrl+F, sprawdź czy zmienna jeszcze gdzieś dalej w tym pliku występuje, jeśli nie to *sayonara*. Z takim podejściem się tylko raz sparzyłem – zmienne od daty (dzień i godzina osobno) są generowane w head.php, jak je scaliłem w jedną i wywaliłem pośrednie to rozwaliło daty wpisywane do bazy danych w dzienniku, na forum nie wyświetlało „N" poprawnie itd. Co nie oznacza, że pomysł jest zły – po prostu za dużo plików do zmiany by było.
- sztuczne rozgraniczanie pewnych rzeczy – przykład pochodzi ze starego includes/battle.php, a poprawka jest moja (w związku z tym, że Eldurvin przerabiał walkę, będzie ona raczej nieaktualna). W każdym razie chodzi o naliczanie obrony gracza w zależności od klasy postaci:

```
if ($defender['clas'] == 'Wojownik' || $defender['clas'] == 'Barbarzyńca')
{
	$epower = ($arrDeequip[3][2] + $defender['level'] + $defender['cond'] +
$arrDeequip[2][2] + $arrDeequip[4][2] + $arrDeequip[5][2]);
	$unik = ($unik + $defender['level']);
	if ($arrAtequip[1][0])
	{
		$unik = $unik * 2;
	}
}
if ($defender['clas'] == 'Rzemieślnik' || $defender['clas'] == 'Złodziej')
{
	$epower = ($arrDeequip[3][2] + $defender['cond'] + $arrDeequip[2][2] +
$arrDeequip[4][2] + $arrDeequip[5][2]);
	if ($arrAtequip[1][0])
	{
		$unik = $unik * 2;
	}
}
if ($defender['clas'] == 'Mag')
{
	if ($defender['mana'] <= 0)
	{
		$epower = ($arrDeequip[3][2] + $defender['cond'] + $arrDeequip[2][2]
+ $arrDeequip[4][2] + $arrDeequip[5][2]);
	}
	else
	{
		$eczarobr = ($defender['wisdom'] * $def_dspell -> fields['obr']) -
(($def_dspell -> fields['obr'] * $defender['wisdom']) * ($arrDeequip[3][4] /
100));
		if ($arrDeequip[2][0])
		{
		    $eczarobr = ($eczarobr - (($def_dspell -> fields['obr'] *
$defender['wisdom']) * ($arrDeequip[2][4] / 100)));
		    if ($eczarobr < 0)
		    {
		        $eczarobr = 0;
```

```
                }
            }
            if ($arrDeequip[4][0])
            {
                $eczarobr = ($eczarobr - (($def_dspell -> fields['obr'] *
$defender['wisdom']) * ($arrDeequip[4][4] / 100)));
                if ($eczarobr < 0)
                {
                    $eczarobr = 0;
                }
            }
            if ($arrDeequip[5][0])
            {
                $eczarobr = ($eczarobr - (($def_dspell -> fields['obr'] *
$defender['wisdom']) * ($arrDeequip[5][4] / 100)));
                if ($eczarobr < 0)
                {
                    $eczarobr = 0;
                }
            }
            if ($arrDeequip[7][0])
            {
                $eczarobr = ($eczarobr + (($arrDeequip[7][2] / 100) *
$eczarobr));
            }
            $epower = ($arrDeequip[3][2] + $eczarobr + $defender['cond'] +
$arrDeequip[2][2] + $arrDeequip[4][2] + $arrDeequip[5][2]);
        }
        if ($arrAtequip[1][0])
        {
            $unik = $unik * 2;
        }
}
if ($unik < 1)
{
    $unik = 1;
}
```

Straszne krzaki, w których nie widać tak naprawdę o co chodzi. A wystarczy zauważyć, że <u>każdemu</u> się dolicza ten jakiś bonus z przedmiotów (nawet nieistotne z jakich!) i wytrzymałość. Wojownikom/barbarzyńcom dokładamy bonus za poziom i już znika `if` o rzemieślnikach i złodziejach. Trochę poprawić tego maga i już kod się zaczyna mieścić na jednym ekranie:

```
$epower = $arrDeequip[3][2] + $defender['cond'] + $arrDeequip[2][2] +
$arrDeequip[4][2] + $arrDeequip[5][2]; // niezależnie od klasy postaci
if ($defender['clas'] == 'Wojownik' || $defender['clas'] == 'Barbarzyńca')
{
      $epower += $defender['level'];
      $unik += $defender['level'];
}
if ($defender['clas'] == 'Mag' && $defender['mana'] > 0)
      if ($defender['mana'] > 0)
      {
            $intTemp = $defender['wisdom'] * $def_dspell -> fields['obr'];
            $eczarobr = $intTemp *  (1 - $arrDeequip[3][4] / 100);
            if ($arrDeequip[2][0])
                  $eczarobr -= $intTemp *  (1 - $arrDeequip[2][4] / 100);
            if ($arrDeequip[4][0])
                  $eczarobr -= $intTemp *  (1 - $arrDeequip[4][4] / 100);
```

```
            if ($arrDeequip[5][0])
                  $eczarobr -= $intTemp *  (1 - $arrDeequip[5][4] / 100);
            $eczarobr = max($eczarobr, 0);
            if ($arrDeequip[7][0])
            {
                  $intN = 6 - (int)($arrDeequip[7][4] / 20);
                  $eczarobr += (10 / $intN) * $defender['level'] * rand(1,
$intN);
            }
            $epower += $eczarobr;
      }
      if ($arrAtequip[1][0])
            $unik *= 2;
      $unik = max($unik, 1);
```

Najlepiej by taka optymalizacja miała miejsce już na etapie projektowania, albo by koder oglądający dany gotowy plik miał wgląd do oryginalnego projektu. Często jest to niemożliwe (kolejne pady forów Rady Królewskiej), ale może akurat gdzieś backup jest. Wydłubywanie takich zależności z gotowego kodu jest trudne (ang. *reverse engineering*, inżynieria odwrotna), choć się opłaca. Bierz się za to tylko jeśli masz dużo czasu, bo na jednym miejscu się nie skończy. Rozhulasz się i cały plik przerobisz (a potem niestety będzie go trzeba dogłębnie przetestować :P).

Inny przykład: funkcja `checkagility()` z includes/funkcje.php (też już przestarzałe?) – chodzi o karę do zręczności w wysokości 1% obrony dawanej przez tarczę, zbroję, nagolenniki:

Vallheru 1.3	Wersja eye, na Oro do lipca 2007

`function checkagility($agility, $armor, $legs, $shield)`
`{`
`  if ($armor > -1)`
`  {`
`  $intArmor = ($agility * ($armor / 100));`
`  }`
`  else`
`  {`
`  $intArmor = $armor;`
`  }`
`  if ($legs > -1)`
`  {`
`  $agi2 = ($agility * ($legs / 100));`
`  }`
`  else`
`  {`
`  $agi2 = $legs;`
`  }`
`  if ($shield > -1)`
`  {`
`  $agi3 = ($agility * ($shield / 100));`
`  }`
`  else`
`  {`
`  $agi3 = $shield;`
	`function checkagility($agility, $armor, $legs, $shield)`
	`{`
	`  $intArmor = ($armor > -1) ? $agility * ($armor / 100) : $armor;`
	`  $agi2 = ($legs > -1) ? $agility * ($legs / 100) : $legs;`
	`  agi3 = ($shield > -1) ? $agility * ($shield / 100) : $shield;`
	
	`  **return $agility - $intArmor - $agi2 - $agi3;**`
	`}`

Vallheru 1.3	Wersja eye, na Oro do lipca 2007

```
    }
    $agi1 = ($agility - $intArmor);
    $newagi = ($agi1 - $agi2);
    $newagility = ($newagi - $agi3);
    return $newagility;
}
```

I wcale nie jest powiedziane że jeszcze się tu czegoś nie wyciśnie:

- usunąć `if`'y całkowicie jeśli założymy że nie ma przedmiotów z karą do obrony
- i zwinąć do jednej linijki (tylko wtedy po co nam funkcja? Koszt skoku przy jej wywołaniu i przekazaniu parametrów jest większy niż zysk z „uwspólnienia" kodu):

```
return $agility - ($armor + $legs + $shield) / 100;
```

Kolejna sprawa to utrudnianie sobie życia:

- wstawiając niepotrzebne nawiasy do wyrażeń – nie zmieniają kolejności wykonywania działań, a tylko mącą,
- śmieszne używanie funkcji z argumentami domyślnymi, np.

```
$intBonus = round($fltBonus, "0");
```

  Nie dość, że zero jest tekstem a nie liczbą, więc PHP musi to dodatkowo konwertować, to jeszcze jest to argument domyślny i jak go ominiemy to właśnie zero zostanie użyte. Wystarczy szybki rzut oka do manuala PHP, `round($zmienna)` i już. `Round`... a `settype('int')` nie łaska?

- Pracowite zapisywanie każdego wyniku (czy to funkcji `rand()`, czy jakiegoś obliczenia) do zmiennej a potem się okazuje że ta zmienna jest używana tylko raz, w jakimś `if`'ie czy jednym wzorze. To po co tą pamięć marnować, lepiej napisać `if (rand(1,100) < 10)` i spokój...
- Jak już przy randomach jesteśmy – wylosowanie liczby kosztuje tyle samo czy to ma zakres 1–2 czy 100–1000000. Zastanów się, czy zamiast `if cośtam` dolosowywać drugą liczbę, nie prościej Ci będzie odpowiednio zmienić zakresy i mieć tylko jedno losowanie. Przykład – choćby pliki wytwórstwa, tzn. kowal.php, lumbermill, jeweller, alchemik...
- pod utrudnianie sobie życia podpada też utrudnianie życia graczowi – więcej w rozdziale 5.7 o efektywnym Smarty'm

Po bodajże **czwarte** – **szanuj RAM**:

- `GetRow`/`GetAll` zamiast kopiowania danych z bazy (opisane w rozdziale 5.3 o ADOdb) – tego

jest naprawdę mnóstwo w kodzie Thindila...

- jeśli skończyłeś pracę z jakąś dużą tablicą, to ją zniszcz funkcją `unset()`. Można `unset`'ować więcej niż jedną zmienną naraz. Można niszczyć wszystkie zmienne, nie tylko tablice, no ale na tych mamy największy zysk pamięci, więc Twoja decyzja czy się chcesz bawić w usuwanie każdej jednej. Jeśli starannie będziesz ciął kod na funkcje i inne logiczne bloki, to zmienne lokalne same znikają po wyjściu z zakresu i wszyscy są zadowoleni.
- Nie deklaruj bzdur jako `DEFINE`, głównie się to tyczy plików językowych. Weźmy takie Posągi, opisy bóstw w Świątyni czy nazwy minerałów i ziół w MK:

```
define('COMBAT_SKILLS', 'Posągi umiejętności bojowych');
define('HIGHEST_SIDEARMS_SKILL', 'Najwyższa Walka bronią białą');
define('HIGHEST_GAME_SHOOTING', 'Najwyższe Strzelectwo');
define('HIGHEST_SPELL_CASTING', 'Najwyższe Rzucanie czarów');
define('HIGHEST_DODGING', 'Najwięcej uników');
define('HIGHEST_LEADERSHIP', 'Najwyższe Dowodzenie');
define('SIDEARMS_SKILL', 'Walka bronią białą');
define('GAME_SHOOTING', 'Strzelectwo');
define('SPELL_CASTING', 'Rzucanie czarów');
//... i w ten deseń niemal cały plik
```

Jedynym sensem istnienia tych stałych jest to, by zostały wsadzone w tabelkę i przypisane do Smarty'ego. To po co tak marnować pamięć i czas procesora na szukanie definicji? Nie lepiej od razu zdefiniować odpowiedniego `array`'a w tym pliku? Jak ktoś będzie chciał go tłumaczyć na angielski to i tak się zorientuje co jest co patrząc na odpowiadającą mu tablicę danych do posągu. Tym bardziej że stałych nie można oddefiniować, `unset`'nąć czy cokolwiek, a po wyświetleniu posągu nie są do niczego potrzebne.

Ostatni element, ale **najważniejszy**: referencje. Zakładam że wiesz co to, jak to działa itd.

Przekazuj przez referencję zmienne do funkcji, które mają zmodyfikować wartość oryginału – szybsze i łatwiejsze od `return`, zwłaszcza gdyby było trzeba zwrócić kilka zmodyfikowanych zmiennych (jest to możliwe jak się ktoś uprze, trzeba zwrócić tablicę).

Zawsze przekazuj przez referencję rzeczy nie będące typami wbudowanymi (int, float, string, bool itd), czyli tablice i obiekty. Dzięki temu unikasz kopiowania (przekazanie przez wartość). W PHP5 tablice i obiekty domyślnie lecą przez referencję (chyba waląc przy okazji jakimś Notice do bugtracka, jeśli w definicji funkcji nie użyto &), ale aktualnie na serwerze mamy czwórkę (a i nie wiadomo na jak biednych hostingach ludzie będą stawiać ten silnik). Poza tym – zwyczajnie nie zaszkodzi programiście jak ma to wyraźnie napisane i się nie musi zastanawiać.

Przez referencję można też przekazać coś Smarty'emu.

```
$smarty -> assign_by_ref('Nazwa', $tablica);
```

Niestety tablice muszą iść pojedynczo, to nie obsługuje składni `assign(array(...))`. Oczywiście ryzykujemy tu, że Smarty nam zmodyfikuje wartości w tablicy. No ale w sumie... skoro wykonujemy `display()` to już nam na tym nie zależy, niech se projektant *.tpl zmienia jak musi. A po display'u oczywiście `unset()`. W razie wątpliwości – obejrzyj na przykład monuments.php.

5.5 Praca z tekstem

Thindil w wielu miejscach używa funkcji `ereg()`. Zasada jest taka:

Tabela 1: Szybkość działania funkcji operujących na stringach. Zauważ że strtok jest wyjątkiem!

Działanie	najszybsza	średnia	najwolniejsza
Znajdź dany znak/tekst w większym tekście	`strpos()`	`preg_match()`	`ereg()`
„Znajdź i zamień“	`str_replace()`	`preg_replace()`	`ereg_replace()`
Tokenizer	`explode()`	`preg_split()`	`strtok()`

`Strpos` i `str_replace` są do prostych operacji – jesli tylko chcemy wiedzieć czy „coś jest” i szukany ciąg jest np. zwykłym wyrazem. Poszperaj po manualu, jest kilka funkcji wykonujących te szybkie przeszukania (w tym np. wersje ignorujące wielkość znaków jak `stripos()`).

„`preg`” i „`ereg`” operują na wyrażeniach regularnych i używa się ich dość podobnie. *Regular expressions* (*regexps*) powinno być tłumaczone raczej jako „wyrażenia rządzące się regułami”, ale niestety tak się już zakorzeniło ;) Pisanie wzorców wyrażeń jest sztuką, więc po szczegóły muszę Cię odesłać do jakiegoś porządnego tutoriala (jako że interesują nas „`preg_X`”, googlaj za „`preg_match`” albo ogólnie stylem wyrażeń zgodnym z POSIX). U nas są używane głównie w includes/bbcode.php - zamiana uśmieszków na obrazki, wstawianie cenzury, zamiana [b] na <b>, *pisanie klimatyczne* itd. Tu warto wspomnieć że `preg_replace` może przyjąć argumenty w postaci tablic. Czyli zamiast 10-ciu wywołań z różnymi wzorcami i „zamień na” możemy mieć jedno odpalenie całego silnika wyrażeń regularnych i niech on sobie bada tablicę tych wyrażeń.

Z innych operacji na stringach warto wymienić `strlen` (długość łańcucha), `wordwrap` (wstawianie spacji), `strtolower` (zamiana na małe litery), `substr_count` (zliczanie ilości wystąpień), `trim` (wycinanie znaków, np zbędnych spacji) i inne („See also” w manualu PHP).

Tokenizer, czyli dzielenie czegoś długiego na kawałki (np. wybieramy separator – spację i tniemy zdania na wyrazy). Thindil nadużywa `explode()` gdy interesuje go jedynie ostatni element (np. sprawdzanie jaki plik rzucił błędem do bugtracka), a w tym momencie mądrzejsze byłoby użycie `preg_match` z wzorcem tego ostatniego członu i znakiem dolara oznaczającym koniec tekstu. „`/[^\/]$/`” załatwia sprawę z bugtrackowym testem i „eleganciej” wygląda.

Ogólnie – pomyśl zanim użyjesz. PHP ma masę gotowych narzędzi, nie trzeba odkrywać Ameryki na nowo (*i pracować w pętli :P*). Rozbudować o konkretną pracę z regexpami?

Na koniec prosta sztuczka na sprawdzenie konkretnej literki w słowie. Weźmy astralne plany, komponenty, mikstury itd. One mają oznaczenia składające się z literki i cyfry, np M2. Oto jak Thindil rozbija to „słowo" na elementy (litera jest potrzebna by stwierdzić co to jest, cyfra mówi, że to np. trzecia pod względem trudności mikstura alchemiczna).

```
// Przykład pochodzi z includes/tribefight.php
// Załóżmy, że $objAstral -> fields['type'] zawiera to „M2".

$arrNames = array('M', 'P', 'R', 'Y', 'C', 'O', 'T', 'J');
$strName = ereg_replace("[0-9]", "", $objAstral -> fields['type']);
//wytnij cyfrę
$intKey = array_search($strName, $arrNames);                    //sprawdź która
to literka
$intKey2 = (int)ereg_replace($arrNames[$intKey], "", $objAstral ->
fields['type']);
// wytnij literkę
```

Nie masz wrażenia, że coś tu na około jest robione? :D

```
$strName = $objAstral -> fields['type']{0};          // pierwszy znak
$intKey2 = (int)$objAstral -> fields['type']{1};     // drugi znak
```

Ech, Thindilek, Thindilek... I nie ma tak, że on nie zna tej sztuczki, stosował ją w klasie Player przy antidotach.

5.6 Praca z tablicą

Tablica jaka jest, kazdy widzi. Deklarowana słowem `array()`, może być zagnieżdżana stosując kolejnego `array`'a. Jeśli każdy element będzie array'em to dostajemy tablicę 2,3,itd-wymiarową; jeśli tylko niektóre (co jest dość rzadkim przypadkiem), to wychodzi nam jakieś mieszane coś.

Indeksacja słowami lub liczbami (że liczbowe są szybsze pisałem przy okazji efektywnego ADOdb w rozdziale 5.3). Przypominam, że indeksowanie liczbowe zaczyna się od zera!

Zliczenie elementów tablicy (tylko najwyższego wymiaru): `count($array)`. Jak ktoś się przyzwyczaił np. z C/C++, może pisać `sizeof()`, to to samo. Jeśli potrzebujemy zliczenia wszystkich elementów we wszystkich wymiarach, trzeba użyć

```
count($array, COUNT_RECURSIVE)(od PHP 4.2).
```

Dostęp odczytu do niezainicjowanej tablicy lub komórki, która nie istnieje, wali Notice'm „`Undefined variable`”/„`Undefined index`” do bugtracka. Rozwiązaniem jest inicjalizacja zmiennej pisząc wcześniej `$arrTablica = array()` i/lub wypełniając odpowiednie pole.

Właśnie poprzez wypełnianie pól poszerza się tablice, wstawia elementy itd. Piszemy `$arrX[7] = 11` i tyle, jeśli tablica miała wcześniej 2 elementy to urośnie. Nie ma tu alokacji pamięci itd. rzeczy znanych z języków o silniejszej kontroli typów. Tu się pojawia taka sztuczka:

```
$arrX[] = 11;
```

przypisze 11 do pierwszej wolnej komórki. Tzn. jeśli ostatnia była `$arrX[7]` no to wstawi pod `$arrX[8]`. Gdy w tablicy są klucze tekstowe, ignoruje je i wstawia najwyższy numer+1 (albo zero jeśli nie było dotychczas liczb).

Wracając do tego Notice'a – zdefiniowanie tablicy przed operowaniem na niej jest poprawne jak najbardziej, ale funkcja `isset()` nam zaczyna zwracać `true`, że zmienna istnieje. A nas by czasem urządzało sprawdzenie czy tablica zawiera jakieś elementy... Rozwiązaniem jest odpalenie... nie, nie `count()` tylko `empty()`. Zawsze to jakieś milisekundy ;) skoro potrzebujemy `true/false` to nie czekajmy aż on się doliczy stu elementów (że co, że głupi przykład? a kowal wystawiający coś na rynek to niby ile chłamu ma w plecaku?). Oczywiście na elementach tablicy już można jak najbardziej `isset()`. Jeśli masz wątpliwości, co zawiera tablica, można ją debugowo wyświetlić:

```
print '<pre>';
print_r($tablica);
print '</pre>';
```

Najczęściej wykonywane w pętli operacje, które powinny być wywołaniem funkcji PHP:

- `in_array` – sprawdza czy element istnieje (tylko `true`/`false`) w tablicy.

- `array_search` – wyszukuje wartość w tablicy, ale zwraca klucz (tzn. indeks liczbowy lub słowny). W przypadku niepowodzenia zwraca `NULL`. To może rodzić pewne problemy, bo normalnie `if` tak samo potraktuje `NULL`, `false` oraz `0` i `''` (a te dwa ostatnie są już calkiem legalnymi indeksami tablicy, nie?). Dlatego powinno się tu stosować operator `===`.

```
Uproszczona tabelka kosztu błogosławieństw do cech (siła, wytka itd). Koszt w
punktach wiary, zależny od rasy postaci.

$arrRaces = array('Człowiek', 'Elf', 'Krasnolud', 'Hobbit', 'Jaszczuroczłek',
'Gnom');
if(!in_array($player -> race, $arrRaces))
    error('Najpierw wybierz rasę!');
// wiersze: człowiek, elf, kraś..., kolumny: siła, wytka,...
$arrFaithCosts = array(array('10', '10', '10', '10', '10', '10'),
                       array('7', '15', '10', '10', '7', '15'),
                       array('15', '7', '10', '10', '15', '7'),
                       array('7', '15', '10', '10', '7', '10'),
                       array('7', '7', '15', '15', '7', '7'),
                       array('10', '15', '10', '15', '15', '10'));
$intRaceKey = array_search($player -> race, $arrRaces); // zwraca 0 dla
człowieka, 3 dla hobbita itd.
print 'Błogosławieństwo do siły kosztuje Cię '.$arrFaithCosts[$intRaceKey][0];
// Przykład mógłby zamiast tego in_array na początku zawierać tylko array_search
ale z porównaniem „czy zwrócono NULL”:
if ($intRaceKey === NULL)
    error();
// Tylko że czy toto zwraca NULL czy false, zależy od wersji PHP (NULL powyżej
4.2), więc tak jest (dla świętego spokoju, na razie) bezpieczniej.
```

- `array_key_exist` – działa trochę odwrotnie do `array_search`, tzn. sprawdza czy dany klucz (indeks liczbowy lub słowny) istnieje w tablicy i ma przypisaną wartość. Różnica między `array_key_exist('indeks', $tablica)` i `isset($tablica['indeks'])` jest taka, że jesli element ma wartość `NULL` (ale jest przypisany), to `isset` da `true` a nasza funkcja `false`.

- `implode()` - stwórz string z elementów tablicy, wstawiając między nie wybrany separator (np. przecinek i spację, albo `'<br />'`)

- `array_walk()` - wywołaj daną funkcję dla każdego elementu tablicy. Nie można dodawać elementów, unset'ować itp. zmieniać tablicy jako takiej. Przydatne tylko przy długich funkcjach, przy krótkich zysk w porównaniu z pętlą nie jest zbyt wielki (więcej czasu marnuje na skok do funkcji niż to warte).

Talice przekazuj do funkcji przez referencję, a niepotrzebne – usuwaj funkcją `unset`!

5.6.1 Foreach

W skrócie: czwarty typ pętli, nadaje się tylko do tablic ale wypas nieziemski ;)

Tradycyjna pętla `for` czy `while` z indeksem jest fajna do momentu gdy:

- trafimy na tablicę, której kluczami wprawdzie są liczby, ale np. z dziurami: 0,1,2,3,7,
- trafimy na tablicę asocjacyjną (kluczami są teksty).

Pierwsze możnaby jakoś rozwiązać wstawiając `if (isset($tab[$i]))` albo używając czegoś w stylu `array_key_exists()`, ale to takie biedne jakieś. Drugie to zupełna porażka, bo potrzebowalibyśmy jakiejś dodatkowej tablicy z tymi tekstami:

```
$arrKeys = array('a', 'bb', 'ccd');
$arrX = array( 'a' => 7, 'bb' => -13, 'ccd' = 'apple');
for ($i = 0, $max = count($arrX); $i < $max; ++$i)
    echo 'Key: '.$arrKeys[$i].', value: '.$arrX[$arrKeys[$i]].'<br />';
```

Trochę głupio to wygląda... Oczywiście jest parę funkcji przetwarzających całą tablicę (`array_walk`), można by też kombinować coś naokoło (`array_keys`, `array_pop`, `array_reduce`...). Ale i tak nienaturalne to.

Od PHP 3 pojawiły się funkcyjki operujące na wewnętrznym wskaźniku tablicy. Idea jest taka, że chodzenie po niej będzie szybsze od ręcznego indeksowania z zewnątrz przez $i++, a dodatkowo rozwiązuje nasze problemy (można przejechać się po całej tablicy nie przejmując się czy jesteśmy na elemencie `$tab[5]` czy `$tab['Lepper']`). Te funkcje to: `current()`, `next()`, `prev()`, `reset()`, `end()` oraz `each()`. Nawet to sprytne było (i jest, polecam zajrzenie do dokumentacji, żeby kolejną rzecz łatwiej zrozumieć).

Programiści domagali się wygodniejszej wersji tych funkcji i tak od PHP 4 dostępna jest instrukcja foreach, znana choćby z Perla czy C#. Są dwie wersje składni i obie przydatne:

```
foreach ($tablica as $wartość)
    rób coś
foreach ($tablica as $klucz => $wartość)
    rób coś
```

Nie ma `count()`, nie ma indeksu tablicy. Samo się martwi o przesuwanie się. W każdym obrocie to, co w normalnej pętli byłoby dostępne jako `$tablica[$i]`, będzie się znajdować pod zmienną `$wartość`. A indeks/klucz, jeśli jest potrzebny, to pod `$klucz`. Czyli nasz przykład wyglądałby tak:

```
foreach ($arrX as $key => $value)
    echo 'Key: '.$key.', value: '.$value.'<br />';
```

Można dowolnie nazwać te zmienne stojące w miejscach `$klucz` i `$wartość`.

Wady:

- pracuje na kopii tablicy, więc nie da się przypisać `$wartość = 3;` (tzn da się, ale przypisze do kopii, a nie oryginału i po opuszczeniu pętli zmian nie będzie widać). To jest raczej do odczytu elementów (ale to tak naprawdę niewiele nam zmienia, zwykle czytamy czy to dane z bazy, czy katalogi, czy inne tablice i robimy jakieś obliczenia na ich wartościach),
- nie można w trakcie działania zmienić tablicy (usunąć element, dodać, transponować macierz itd.), bo jeździ tym wewnętrznym indeksem. Ale słowa kluczowe jak `break` czy `continue` normalnie działają,
- nie można użyć @ do ukrycia komunikatów błędów.

Zalety:

- prosta składnia, bez `count()`, warunku końca pętli, `$i++` itp. dodatków zaciemniających to, co naprawdę chcemy zrobić,
- **najszybsza** pętla po tablicy, nieznacznie przebijająca porządnie napisanego for'a! To przez to użycie wewnętrznego indeksu,
- pozwala wprowadzić trochę uniwersalności do kodu. Przestaje być istotny np. *fetch mode* z bazy danych, bo siódma wartość to wartość, a czy użyto numerka (indeksu) czy napisu (klucza – nazwy kolumny w tabelce bazy danych) by sie do niej dobrać, to nieważne. Przynajmniej tak długo jak praca „asoscjacyjna" będzie używać elementów w takiej kolejności w jakiej są wypisane w SELECT'cie.

Z poważniejszych miejsc w jakich mi się przydała dotychczas to ekwipowanie weterana. Mam masę zmiennych w `$_POST`, klucze mówią mi o jaką część ekwipunku chodzi, wartości to ID przedmiotów w tabelce equipment. Każdy element muszę sprawdzić czy należy do gracza, czy nie jest założony albo w szafie w domku, wyliczyć jego moc, usunąć albo zmniejszyć licznik elementów... Aż się prosi o pętlę. Ale co z tego, że wiem jakie słowa zawiera ta `$_POST`, skoro musiałbym skorzystać z pomocniczej tablicy słów tak jak przykładzik napisałem na początku tego rozdziału. A z foreach dużo bardziej elegancko to wyszło.

Wyjątkowo nie daję przykładu „prawdziwego użycia". Jest tego mnóstwo w kodzie, poszperaj!

5.7 Elegancki Smarty

Przypisywanie w *.php do `$smarty` komunikatów z pliku językowego jest brakiem szacunku do graczy – jest to durne kopiowanie żrące pamięć. Stała `NAZWA_STAŁEJ` z languages/pl/*.php jest w pliku *.tpl widoczna jako `$smarty.const.NAZWA_STAŁEJ`.

Podobnie z elementami tablic globalnych takich jak `$_GET`, `$_POST`, `$_COOKIE`, `$_SESSION`, `$_SERVER` – na dole każdego pliku *.php mamy coś takiego:

```
if (!isset($_GET['step']))
{
    $_GET['step'] = '';
}

if (!isset($_GET['action']))
{
    $_GET['action'] = '';
}

/**
* Assign variables and display page
*/
$smarty -> assign(array('View' => $_GET['view'],
                        'Step' => $_GET['step'],
                        'Action' => $_GET['action']));
$smarty -> display('admin.tpl');
```

A w *.tpl `{if $View = 'cośtam'}`... Smarty widzi wszystkie wymienione wcześniej tablice, wystarczy sprawdzać element `$smarty.get.view`, `$smarty.get.step` dla uzyskania identycznego efektu. Ale to taka kosmetyka.

O `{strip}` już pisałem – zysk z usunięcia zbędnych spacji czasem doprowadza do odchudzenia pliku wynikowego o 60% (zapisz sobie na dysk html z monuments.php wygenerowany bez strip'a. Tylko pamiętaj o wywaleniu wcześniej pliku z katalogu „cache”. Ostatnio miałem 24 a 56 kB.) Ogólnie – zależy od zagłębienia pętli w templejcie i ilości wcięć, jakie można bezpiecznie wywalić.

Kolejna rzecz jaka mnie irytuje to kombinowanie na siłę z „action” formularzy. Thindil uwielbia dorzucać jakieś „step” do paska adresu, a jak mu brakuje to step2, step3... (nie mów, że nigdy Cię nie wkurzyło ile się trzeba naklikać by dotrzeć do odpoczynku w domku). Weźmy przykład z wpłatą/wypłatą kasy w skarbcu Strażnicy (jest toto tak rozwlekłe że pewnie nic nie będzie widać. Najlepiej otwórz odpowiedni plik Vallheru 1.3):

outposts.php	outposts.tpl

/**
* Add gold to outpost or take it from outpost
*/
if (isset ($_GET['view']) && $_GET['view'] == 'gold')
{
$smarty -> assign(array("Gold" => $out -> fields['gold'],
"Goldinfo" => GOLD_INFO,
"Goldcoins" => GOLD_COINS,
"Atake" => A_TAKE,
"Aadd" => A_ADD,
"Fromout" => FROM_OUT,
"Toout" => TO_OUT));
/**
* Get gold from outpost
*/
if (**isset ($_GET['step']) && $_GET['step'] == 'player'**)
{
if (!isset($_POST['zeton']))
{
error(HOW_MANY);
}
if (!ereg("^[1-9][0-9]*$", $_POST['zeton']))
{
error (ERROR);
}
if ($_POST['zeton'] > $out -> fields['gold'])
{
error (NO_MONEY);
}
$zmiana = floor($_POST['zeton'] / 2);
$zmiana = (int)$zmiana;
$db -> Execute("UPDATE players SET credits=credits+".$zmiana." WHERE id=".$player -> id);
$db -> Execute("UPDATE outposts SET gold=gold-".$_POST['zeton']." WHERE owner=".$player -> id);
$smarty -> assign ("Message", YOU_CHANGE.$_POST['zeton'].GOLD_ON.$zmiana.GOLD_ON2);
}
/**
* Add gold to outpost
*/
if (**isset ($_GET['step']) && $_GET['step'] == 'outpost'**)
{
if (!isset($_POST['sztuki']))
{
error(HOW_MANY);
}
if (!ereg("^[1-9][0-9]*$", $_POST['sztuki']))
{
	{if $View == "gold"}
	{$Goldinfo} {$Gold} {$Goldcoins}.
	
	<form method="post" **action="outposts.php?view=gold&step=player**">
	<input type="submit" value="{$Atake}" /> <input type="text" name="zeton" value="0" />
	{$Fromout}.</form>
	<form method="post" **action="outposts.php?view=gold&step=outpost**">
	<input type="submit" value="{$Aadd}" /> <input type="text" name="sztuki" value="0" /> {$Toout}.</form>
	{$Message}
	{/if}

outposts.php	outposts.tpl

`error (ERROR);`
`}`
`if ($_POST['sztuki'] > $player -> credits)`
`{`
`error (NO_MONEY);`
`}`
`$db -> Execute("UPDATE players SET credits=credits-".$_POST['sztuki']." WHERE id=".$player -> id);`
`$db -> Execute("UPDATE outposts SET gold=gold+".$_POST['sztuki']." WHERE owner=".$player -> id);`
`$smarty -> assign ("Message", YOU_ADD.$_POST['sztuki'].GOLD_TO);`
`}`
`}`	

W templejcie widać dwa zmodyfikowane linki w action, które są badane na `isset()` w *.php i decydują czy wpłata czy wypłata. W sumie po co? Jeśli zostawimy linki jak są (**`outposts.php?view=gold`**) i zbadamy `isset()`'em nazwy z formularzy, tzn. `$_POST['zeton']` lub `$_POST['sztuki']`, to nam odpadnie trochę roboty. Jeśli żaden nie jest ustawiony, to albo gracz sobie wyświetlił skarbiec i (jeszcze) nic nie robił, albo próbował być sprytny, grzebał w kodzie HTML formularza. W obu przypadkach można „olać", nie wykonując żadnej operacji.

Resztę optymalizacji będziemy wykonywać w *.php, to tu tylko jeszcze jedna uwaga do *.tpl – skoro zostawiamy **„action" prowadzący do tej samej strony**, to **można wpisywać pusty!** Przeglądarka zrozumie to jako „wyślij dane na tą samą stronę". I spokój z kombinowaniem na siłę.

Okiej, pozbyliśmy się durnych `$_GET['step']`. Przy okazji można wywalić ten `if` z errorem `HOW_MANY`. Kod (przykładowo dla wpłaty) zaczyna wyglądać tak:

```
/**
 * Add gold to outpost
 */
if (isset($_POST['sztuki'])
{
    if (!ereg("^[1-9][0-9]*$", $_POST['sztuki']))
    {
        error (ERROR);
    }
    if ($_POST['sztuki'] > $player -> credits)
    {
        error (NO_MONEY);
    }
    $db -> Execute("UPDATE players SET credits=credits-".$_POST['sztuki']."
WHERE id=".$player -> id);
    $db -> Execute("UPDATE outposts SET gold=gold+".$_POST['sztuki']." WHERE
owner=".$player -> id);
    $smarty -> assign ("Message", YOU_ADD.$_POST['sztuki'].GOLD_TO);
}
```

`ereg()`, czyli sprawdzenie czy gracz wpisał liczbę, zwalimy na nowo powstałą funkcję z pliku includes/security.php. Ta funkcja jest bardziej tolerancyjna od tego `ereg`'a, jak również od „śmiech na sali" znanej jako `integercheck()`.

Następnie warto pójść graczowi na rękę i nie walić tym errorem `NO_MONEY` po oczach. Czy tak trudno przyjąć, że jeśli wpisał za dużo, to chce wpłacić całość? Gracz może docisnąć klawisz i wpisać 99999999999999, a my mu to przetworzymy (część tego również załatwia nowa funkcyjka, część już jest schowana w kodzie klasy opisującej strażnicę).

Na koniec trochę uwspólniłem kod wpłat i wypłat, dokleiłem to do nowo stworzonej klasy opisującej strażnicę i rozdzieliłem część „obsługa formularza" od faktycznej wpłaty/wypłaty:

```
/**
* Add gold to outpost or take it from outpost
*/
if (isset ($_GET['view']) && $_GET['view'] == 'gold')//ogólnie Skarbiec
{
    if (isset($_POST['zeton']))     // zmienna z formularza wypłat, więc wiadomo
    {
        if (!uint32($_POST['zeton'])) // funkcja z includes/security.php
            error(ERROR);
        $smarty -> assign ('Message', YOU_CHANGE.$_POST['zeton'].GOLD_ON.($out
-> goldBalance(-$_POST['zeton'])).GOLD_ON2);
    }
    if (isset($_POST['sztuki']))
    {
        if (!uint32($_POST['sztuki']))
            error(ERROR);
        $player -> credits -= $out -> goldBalance($_POST['sztuki'], $player ->
credits);
        $smarty -> assign ('Message', YOU_ADD.$_POST['sztuki'].GOLD_TO);
    }
    $smarty -> assign(array('Treasury' => $out -> gold,
                            'GoldInHand' => $player -> credits));
}
```

Zauważ że assign do Smarty'ego jest pchnięty na sam koniec, a najpierw obsługujemy formularz!

Plik klasy. Funkcja dość dziwnie wygląda, ale w sumie prosta jest (przede wsiem chodziło o trzymanie zapytań do bazy danych w jednym miejscu). Drugi argument to złoto gracza – bez znaczenia przy wypłatach, ale pomocne przy wpłacie.

```
public function goldBalance($intAmount, $intPlayerGold = 0)
{
    if ($intAmount > 0)    // Deposit gold to outpost.
    {
       $intAmount = min($intAmount, $intPlayerGold);// jesli za dużo, to
przytnij
       $intGold = $intAmount;
       $this -> gold += $intAmount;
       $deposit = true;
    }
```

```
    else    // Withdraw gold from outpost with 50% penalty.
    {
        $intAmount *= -1;
        if ($intAmount > $this -> gold)   // jesli za dużo, to przytnij
            $intAmount = $this -> gold;
        $intGold = floor($intAmount / 2);
        $this -> gold -= $intAmount;
        $deposit = false;
    }
    $this -> db -> Execute('UPDATE `players` SET
`credits`=`credits`'.($deposit ? '-' : '+').$intGold.' WHERE `id`='.$this ->
ownerId);
    $this -> db -> Execute('UPDATE `outposts` SET `gold`='.$this -> gold.' WHERE
`owner`='.$this -> ownerId);
    return $intGold;
}
```

Zwracamy kwotę operacji – przy wpłacie bez znaczenia, przy wypłacie warto wiedzieć ile dostaliśmy po podzieleniu przez 2. Assign do smarty'ego jest wykonywany po wszystkim, więc wyświetlimy nowy stan konta – **gracz nie musi odświeżać strony**!

Na koniec kawałek *.tpl (bez sliderów, AJAX itd, bo czeka na wypracowanie przez Klausa i Aranwe jakiegoś standardu):

```
{if $View == 'gold'}
    {$smarty.const.GOLD_INFO} <b>{$Treasury}</b> {$smarty.const.GOLD_COINS}.<br
/><br />
    <form method="post" action=""> {*Pusty!*}
        <input type="submit" value="{$smarty.const.A_TAKE}" />
        <input type="text" id="treasury" name="zeton" value="{$Treasury}"/>
{$smarty.const.FROM_OUT}.
    </form>
    <form method="post" action="">
        <input type="submit" value="{$smarty.const.A_ADD}" />
        <input type="text" id="goldinhand" name="sztuki" value="{$GoldInHand}"
/> {$smarty.const.TO_OUT}.
    </form>
    <p>{$Message}</p>
{/if}
```

Wystarczyło trochę poprzestawiać :)

Dopisać o pluginach do Smarty'ego.

5.8 Cache

Caching polega na przechowywaniu sobie lokalnej kopii pewnych danych, które były ostatnio używane. Pracuje się na tej „bliskiej" danej ile się da, jak najrzadziej wysyłając wynik na zewnątrz (do RAM-u, do pliku, do bazy danych, do osoby oglądającej stronę).

W ten czy inny sposób „keszowanie" już przerabialiśmy np. stosując zmienne lokalne. Pobieramy pola z bazy danych i pracujemy na kopii, zapisując dopiero gotowe wyniki, a nie pisząc i czytając co chwilę z bazy. Samo pojęcie powinieneś intuicyjnie rozumieć, bo spotkałeś się z tym już nie raz:

- co to jest rejestr procesora niekoniecznie musisz wiedzieć, ale jeśli składałeś komputer i porównywałeś procesory lub karty graficzne, to podejrzewam, że na pytanie „co to kurna jest to L2 cache" ktoś Ci poradził: „a taka pamięć podręczna procka. Nieważne jakim cudem jest dużo szybsza od RAMu, ważne że im więcej tym lepiej",
- najszybciej działa procesor i okolice, potem jest RAM, najwolniejsze są dyski twarde i CD/DVD. W systemie operacyjnym mamy pamięć wirtualną i częściej używane programy zwykle są w RAMie „pod ręką", a reszta wylatuje na dysk.

Starczy tego wstępniaka. Nas będzie interesować przechowanie ciężkiej krwawicy obliczeniowej serwera w przypadku danych, które rzadko się zmieniają. Zostaną policzone raz, a potem następne odwołanie oszczędzi roboty. Sposobów realizacji tego jest zaskakująco dużo:

- **keszowanie** kodu **plików php** – przypomina fazę kompilacji normalnego programu – czasem kompilacja trwa długo ale wynikiem jest szybki *.exe. W PHP tak jakby za każdym razem pracujemy na źródle pliku i ciągle musi on być kompilowany do formy przejściowej (*opcodes*, Zend Engine itd.), wykonywany i dopiero wyniki wysyłane userowi. A przecież pliki się rzadko zmieniają. Efekt ich działania – owszem, dla każdego gracza jest inny. Ale kod źródłowy jest taki sam całymi tygodniami, póki nie implementujemy jakiegoś nowego projektu. Stosuje się więc programiki (moduły do PHP) zapamiętujące np. w RAMie tą pośrednią formę pliku i jeśli oryginał nie uległ zmianie, to przeskakuje etap „kompilacji". Zamierzam zainstalować na serwerze eAccelerator, inne znane to oczywiście Zend Accelerator (płatny ;-( ), Alternative PHP Cache czy AfterBurner Cache. (Stronka eAcceleratora twierdzi, że przyspiesza strony oparte na Smarty'm nawet dziesięciokrotnie. Ładnie, pięknie, zobaczymy, szkoda że dawno nie wyszła jakaś nowsza wersja).
- **buforowanie wyjścia** i kompresja GZIP – coś już o tym pitoliłem przy rozpisce różnic między

`echo` i `print`. Po szczegóły googlać za *output buffering*,

- **cache** wyników **zapytań** do bazy danych – pewnie intuicyjnie czujesz że rzadko zmieniające się dane („ostatnio zarejestrował się") są idealnym kandydatem. ADODB to obsługuje, ADODBLite na chwilę obecną nie, nie wiem jak z PDO jest. Po co pobierać z bazy, jeśli można kazać skeszować i czyścić cache w momencie aktywacji nowego gracza. I już 1 zapytanie mniej na każdej stronie gry! Na razie się tym nie zajmujemy, ma to sens dopiero gdy serwer bazy danych mieli w stosunku do serwera www, a u nas czasy PHP są jak widać, 10-20 razy dłuższe.
- **cache wynikowego HTML**'a, jeśli cała strona się niewiele zmienia. Posągi i ranking (pozwalamy się zdeaktualizować cache po godzinie) to idealny przykład. Kolejne to updates.php czy zawartość sklepów – przecież to się zmienia tylko w momencie gdy Król w panelu doda wieść czy przedmiot. Większość szanujących się systemów szablonów to obsługuje (**Smarty** i OPT też), jak nie ma to można skorzystać z PEAR Cache lub nowszego Cache Lite. Zresztą Smarty nawet niepytany przetwarza *.tpl do pewnej roboczej postaci żeby potem szybciej (he, he) działać – patrz zawartość templates_c.
- bardziej ezoteryczne rozważania (forkująca konfiguracja Apache kontra wątki, kompilacja PHP ze wsparciem dla pamięci współdzielonej) sobie podarujemy.

Tak się zastanawiam czy i ile byśmy oszczędzili pchając np. złoto, energię i poziom postaci do zmiennych sesyjnych i zapisując do bazy przy wylogowaniu. Byłby problem z aktualizacją tego np. przy okradaniu delikwenta (jak się dobrać do nie swojego $_SESSION?), ale może coś się da?

5.8.1 Smarty

Zanim zaczniemy, upewnij się że katalog „cache” istnieje i PHP może do niego pisać.

Ogólny schemat pliku z cache (pogrubiłem wszystko czego nie ma w „surowym” pliku-bazie):

```
<?php

$title = 'Przykładowy plik';
require_once('includes/head.php');
require_once('languages/'.$player -> lang.'/X.php');

if ($player -> location != 'Altara')                  // jakaś kontrola błędów
    error (ERROR);

$smarty -> cache_dir = 'cache/';                      // wskaż katalog
$smarty -> caching = 1;                               // i włącz cache
if (!$smarty -> is_cached('X.tpl'))     // nie ma w cache lub jest przestarzałe?
{
    // Pobrania z bazy danych, obliczenia itp. normalne „mięsko” pliku.
    // ...
    $smarty -> assign_by_ref('Array', $arrX);
    $smarty -> assign('Y', $intY);
}
$smarty -> display ('X.tpl');             // jeśli to możliwe, pobierze z cache
if (isset($arrX))                               // wyczyść pamięć po display'u
    unset($arrX);
$smarty -> caching = 0;                         // wyłącz cache przed foot'em
require('includes/foot.php');                   // bo inaczej też go skeszuje
?>
```

Proste? W katalogu cache powstanie plik o głupiej nazwie (podobnej do tych z templates_c). Zawiera on w zasadzie wynikowy normalny HTML, tylko że na początku jest trochę danych np. o czasie ważności. Odśwież stronę, w stopce powinien spaść czas działania i ilość zapytań.

1. Domyślny czas to godzina. Jeśli życzymy sobie inny, trzeba podać przed `display()` czas w sekundach:

```
$smarty->cache_lifetime = 36000;    // 10 godzin
```

2. Nie można zmienić czasu ważności już display'niętego pliku tą instrukcją. Pozostaje ręczne usunięcie pliku z katalogu albo wywołanie:

```
$smarty->clear_cache('X.tpl');
$smarty->clear_all_cache();         // czyści cały katalog
```

3. Do celów debugowania skryptu można ustawić `$smarty -> compile_check = true;` albo `$force_compile = true;`, ale nie polecam. Najprościej wyłączyć keszowanie:

```
 $smarty -> caching = 0;
```

4. Załóżmy, że mamy nasz plik X z cache godzinnym, a po nim mocno uproszczony foot.php. Foot zawiera tylko wyświetlenie listy graczy, bez overliba, licznika zapytań, zegara itd.

Korzystne byłoby ustawienie keszowania tego foot'a choćby na 2 minuty (lista zalogowanych graczy = gracze aktywni w ciągu ostatnich 3ch minut). Do kompletu `clear_cache()` jak ktoś się zaloguje/wyloguje i jest bomba. Dobra, ale mamy tylko jedną zmienną `cache_lifetime` i jej przestawienie nic nie da! Żeby każdy plik miał własny czas, trzeba ustawić na początku

```
$smarty -> caching = 2; // zamiast 1
```

i dopiero teraz można tym *lifetime* manipulować ustawiając każdemu plikowi inny czas przed wywołaniem `display()`.

5. Jeśli potrzebujemy skeszowania całego *.tpl oprócz określonych części, można użyć tagów `{dynamic}` albo `{insert}`'nąć inny szablon. Jeszcze mi się nie zdarzyło zastosować, więc odsyłam do dokumentacji. Podejrzewam że po prostu trzeba umieścić część kodu (zapytania do bazy, obliczenia, assign'y itp.) poza tym if'em, a przed wyświetleniem.

To tyle z podstaw. Naprawdę zajrzyj do posągów czy rankingu zanim pójdziesz dalej, bo się troszkę komplikuje.

Podany sposób nadaje się niestety wyłącznie do plików... nazwijmy je „read-only”. Posągi, ranking - ok (ciekawostka: Thindil keszuje pliki wyświetlane poprzez source.php). Ale Wieści, artykuły w gazetce czy np. sklep zbrojmistrza niestety nie:

- bo potrzebujemy zmienną ilość komentarzy (choć to by się dało obejść albo tym `{dynamic}` albo `clear_cache()` po dodaniu komentarza),
- bo te strony mają podstrony i cache nałożony na całe „Wieści” sprawi że zobaczymy to samo po kliknięciu na link komentarza (a w gazetce to zupełna paranoja, tylko spis treści ☺).

Nie ma co się poddawać, bo te stronki faktycznie rzadko się zmieniają i wypadałoby jednak coś zrobić. Potrzebujemy cache wybiórczego.

Funkcje `display()`, `is_cached()` oraz `clear_cache()` przyjmują więcej niż jeden argument. Drugim argumentem jest tzw. `$cache_id`. Wszystko jedno, jakie sobie wymyślimy, byle do każdej z nich podać takie samo. Wybrane „id” się pojawia na początku tej głupawej nazwy pliku w katalogu cache. Jeszcze są grupy cache, ale to już dla chętnych do doczytania.

Przykład omówi to, co niedawno zrobiłem z Magazynem Królewskim. Uznałem go za kandydata do keszowania, bo wielu graczy tylko do niego zagląda sprawdzić ceny, a potem idzie na rynek. Bardzo dobrze, niech idą, ale ta ilość zapytań w stopce po wejściu do warehouse.php trochę powala na kolana. A dane się dość rzadko zmieniają. Dodatkowo chciałem zrobić wykresy historii

cen – też na pewno chętnie by były przeglądane a nie powinny obciążać serwera. Z drugiej strony nie może być jak z posągami, że „aktualizacja za 15 minut" - ilość surowców musi odpowiadać prawdzie.

No to wymyśliłem tak:

- strona główna ma być keszowana,
- strony „kup/sprzedaj" nie, bo to bez sensu – i tak za każdym razem będą inne bo zawierają ile kto ma danego surowca albo ile jest w MK. Za to po udanej transakcji powinny czyścić cache strony głównej,
- będzie 26 stron z historią minerału/zioła, każda powinna mieć własne cache. Po transakcji nie trzeba niszczyć wszystkich stron tylko tą jedną, odpowiadającą danemu minerałowi/ziołu,
- reset główny nalicza nowe ceny i przesuwa dni na wykresach, więc powinien niszczyć wszystko.

To był całkiem fajny plan i jedyne na co się nadziałem potem to fakt, że w Altarze i Ardulith są różne teksty na stronie głównej, więc na nią ostatecznie 2 cache poszły. Strona główna będzie miała `$cache_id` takie, jakie miasto ma gracz wpisane w `$player -> location`. Stronki z historią są mało odkrywcze, ich id mają postać od „`h|0`" do „`h|26`".

Wyszedł mi taki szablon strony:

(trzeba pamiętać że to poglądowe jest, nie wstawiłem kontroli błędów. `$cache_id` jest używane do stworzenia pliku, chamski gracz i bardzo długa nazwa → mogą się dziać cuda-wianki, przepełnienie bufora, próby wyjścia poza katalog, nadpisania czegoś itd. Kontrola musi być!).

```php
<?php
require_once('includes/head.php');
require_once('languages/'.$player -> lang.'/warehouse.php');
if ($player -> location != 'Altara' && $player -> location != 'Ardulith')
    error (ERROR);

if (!isset($_GET['action']) || $_GET['action']=='history')
{
    $smarty -> cache_dir = 'cache/';
    $smarty -> caching = 1;
}

/**
* Strona główna
*/
if (!isset($_GET['action']))
{
    if (!$smarty -> is_cached('warehouse.tpl', $player -> location))
    {
        // Pobrania z bazy, obliczenia i $smarty -> assign'y.
    }
    $smarty -> display('warehouse.tpl', $player -> location);
}

/**
* Kup/sprzedaj zioła i minerały
*/
if (isset($_GET['action']) && ($_GET['action'] == 'sell' || $_GET['action'] ==
'buy'))
{
    // Normalny kod wyświetlający „ile chcesz kupić/sprzedać”.
    // ...
    if (isset($_POST['amount']) && strictInt($_POST['amount']))
    {
        // Skoro tu wlazł, to gracz wysłał formularz z zakupami.
        // Jeśli coś jest nie tak to tu walę error'em po oczach.
        // ...
        // Tu już transakcja doszła do skutku, trzeba poczyścić cache.
        $smarty -> clear_cache('warehouse.tpl', 'Altara');
        $smarty -> clear_cache('warehouse.tpl', 'Ardulith');
        $smarty -> clear_cache('warehouse.tpl', 'h|'.$_GET['item']);
    }
    $smarty -> assign(...));
    $smarty -> display('warehouse.tpl'); // normalny, bez drugiego argumentu
}

/**
* Historia zmian wybranego surowca. Idea ta sama co na stronie głównej.
*/
if (isset($_GET['action']) && $_GET['action'] == 'history' &&
isset($_GET['item']))
{
    if (!$smarty -> is_cached('warehouse.tpl', 'h|'.$_GET['item']))
    {
        // ble ble ble
    }
    $smarty -> display('warehouse.tpl', 'h|'.$_GET['item']);
}

$smarty -> caching = 0;
require_once('includes/foot.php');
```

```
?>
```

Mi się to wydaje dość proste i szybkie we wstawianiu. Oczywiście trzeba pamiętać że to przykład, a nie wyrocznia. W innym pliku inne rzeczy mogą wymagać cache. Ale w ten deseń można dużo plików przepisać i mieć zysk małym kosztem, bo co to jest trochę ifów wstawić. Linijka

```
// Jeśli coś jest nie tak to tu walę error'em po oczach.
```

ma tą zaletę, że wywołanie `error()` kończy się foot'em i wyjściem z PHP. Czyli omijamy kod czyszczący cache i zostaje po staremu. Natomiast wada jest taka że nie mogłem użyć `error()` do wyświetlenia „Kupiłeś X illani za Y złota". Przypisałem komunikat do Smarty'ego, w tpl-ce sprawdziłem czy jest ustawiony i jesli tak to wyświetlam. W sumie to chyba nawet bardziej logiczne, że ta funkcja faktycznie będzie do sygnalizowania błędów i wyjątkowych sytuacji.

Co jeszcze tu można napisać... Pierwsza osoba która się nadzieje na keszowanie musi trochę dłużej poczekać (zapisanie pliku na dysk). Więc trzeba rozważnie dobierać kandydatów na pliki do keszowania. Ideałem jest, jeśli zawierają dużo obliczeń lub pobrań z bazy, i jest to wykonywane rzadko (a przynajmniej dużo więcej odczytów niż zapisów), a aktualizowane są rzadko.

Jest kilka pliczków które się nadają do pełnego cache jak posągi: FAQ (no, z podstronami), zegar miejski (jeśli wywalić tekst „najbliższy reset za", bo to i tak na każdej widać. Albo te dynamiczne). I to wielodniowe cache nawet mogłoby być :)

Znacznie więcej jest takich, które wymagają cache selektywnego, ale raczej „rządzą się same" tak jak MK – ustawianie i czyszczenie to kwestia przerobienia jednego, dwóch plików:

- Gazetka, Sąd, Biblioteka, Aleja Zasłużonych – wymaga zmiany w nich i wstawienia `clear_cache()` w odpowiednim fragmencie Panelu Admina (nadawanie rang, dodanie darczyńcy),
- wieści, plotki, sklepy (gotowe przedmioty i plany), ankiety – też tylko właściwy plik do przeróbki, akcja admina + reset główny w przypadku plotek i ankiet. Nawet city.php tu pasuje – to byłby zwykły plik jak posągi gdyby nie migające „N" przy nowych ankietach,
- forum zwykłe i klanowe – troszkę więcej roboty, bo król dodaje fora, do tego uprawnienia, czyszczenie i samo pisanie... Myślę że tu sensowność by zależała od aktywności graczy.
- cache tylko fragmentu pliku: na Arenie Walk lista potworów (podział na dwa miasta, dopisanie czyszczenia przy królewskim „dodaj potwora" i już). Lista klanów, opisy, magazyny klanowe, zbrojownie itd.

Cache zależne od ID gracza – a czemu nie? Tylko żeby docelowa grupa nie była zbyt duża bo nam dysk zawali... Na przykład Księga Czarów – tylko dla magów, a lista się rzadko zmienia. A skoro wywaliliśmy z lokacji będącej następcą Avan Tirith znajdowanie czarów, to tylko sklepik z czarami zostaje do dodania `clear_cache()`.

Oczywiście są pliki które się nie nadają, więc trzeba umiar zachować. Arena Walk czy Strażnica są stworzone do tego, by się zmieniać właściwie na każdej podstronie. Tu nawet personalizacja po ID gracza nic nie da. Musielibyśmy co chwilę czyścić cache i wyjdzie mielenie dyskiem. Tak samo view.php – pozornie cache tego wygląda obiecująco, ale obrywamy po dupie choćby przez pole „ostatnio widziany".

Nad keszowaniem notatnika, dziennika i poczty (tylko listy głównej, bo cache listów to samobójstwo) bym się jeszcze zastanowił. Tyle, że to stworzy docelowo tyle plików, ilu graczy jest, to nie wiem jak z miejscem będzie (czyszczenie co reset główny?).

6 Inne

Rzeczy, które nie pasowały do innych rozdziałów. Kwestie dyskusyjne, rozważania i takie tam.

6.1 W3C Validator

Jest przereklamowany, można go też łatwo ogłupić „poprawnym” acz bezsensownym:

```
<a><em><a /></em></a>
```

Na domiar złego trzymamy się wersji Transitional i tak naprawdę nasz XHTML jest g**** warty, bo nie ustawiamy odpowiedniego MIME-Type (http://pornel.net/ (jaka podstrona?), http://www.doktorno.boo.pl/index.php?q=art008). No nic, kiedyś to poprawimy :)

Póki co staramy się trzymać standard chocby w ten sposób, by wszystkie
 mialy ten *slash* na końcu. Spacja jest dość istotna, stare przeglądarki nie lubią tych tagów w innym wypadku. Gwoli przypomnienia: jesli tag jest pusty (np. `br`, `img`), można użyć formy skróconej:

```
<tag></tag> = <tag />
```

Oczywiście Internet Exploder ma na ten temat własne zdanie, spróbuj użyć skróconego tagu `<title />` :D

Jest fajny program badający poprawność XHTML: HTML Tidy dostępny *standalone* i jako plugin do Firefox'a. Z ciekawszych warto jeszcze wymienić Firebug'a -inspekcja Javascript, edytor DOM z podglądem na bieżąco. Warto byś miał zainstalowanego co najmniej Tidy'ego.

Poza tym istnieje plugin Tidy do PHP, powinniśmy rozważyć podpięcie go i naprawę plików po ostatnim `$smarty -> display` a przed Gzipowaniem.

Obecnie zmiana MIME-type nam rozpiździ grę. Nie mamy zamiany encji np. & na & i wszystkie nicki Tallamarów (&&Nailo Mzah&&) wyskoczą jako błędy parsowania. Widziałem to w akcji: znalazłem artykuł z nickiem Nailo w gazetce i zapisałem go z poziomu spisu treści gazety (prawy klik i „zapisz element docelowy jako”, nie wejście do artykułu i „Plik→ Zapisz jako”); w ten sposób dostajemy dość surową stronę, bez folderu z użytymi CSS, obrazkami itd. Otworzyłem w Firefoxie, on rozpoznał DOCTYPE i się chciał zastosować, przeczytać XHTML i jebut. Można też zmienić rozszerzenie na xhtml.

Odpowiednia funkcyjka roboczo wygląda tak (includes/security.php):

```
function outString(&$strText)
{
   $strText = htmlentities(strip_tags($strText), ENT_QUOTES, 'UTF-8');
}
```

Ale tak naprawdę to jeszcze jest kwestia niedomkniętych tagów, a podpięcie parsera XML nam zwooolni... Trudno, na chwilę obecną jesteśmy interpretowani jako HTML z błędami. Są ważniejsze rzeczy, jak wywalanie tagów całkowicie z *.php i po<b>a
anego kodu z *.tpl (rozwinięcie css-ów). 4 lata gra wytrzymała to jeszcze pół roku postoi.

6.2 Personalizacja „klona”

Instalacja gry na serwerze to nie wszystko (przynajmniej obecnie, mam nadzieję, że konfigurowalność gry się zmieni do wersji 2.0 i po to piszę ten rozdział, jako kawałek „rzeczy które nie są zbyt łatwe i intuicyjne, trzebaby kiedyś coś z tym zrobić”). Trzeba trochę poedytować pliki, a czasem i bazę danych. Ten rozdział zawiera najczęstsze problemy i pytania, jakie zadają administratorzy gier opartych na silniku Vallheru. Porady będą skrótowe, dokładniejsze objaśnienia zostały wielokrotnie rozpisane na następujących stronach: (sprawdzić prawa autorskie + ewent. zgody na dołączenie do tego dokumentu, dodać linki):

- wiki „technicznej” projektu Vallheru,
- forum technicznym,
- archiwach http://erythanea.xve.pl,
- archiwach Krainy Fantasy,
- forum Cannibals.

Tam też powinieneś szukać odpowiedzi, a dopiero potem pytać.

Wyjątkiem są zmiany wprowadzone przez Orodlin Team. Radykalna zmiana zarządzania grą, wizji itd. przy przedłużającej się nieobecności Thindila zaoowocowała dużą ilością słabo opisanych zmian między wersją silnika 1.3 i 1.4 (wstawić streszczenie art. Dellasa z gazetki?). Pojawiła się tu dość spora nieciągłość i część rozwiązań omówionych tutaj nie ma żadnego związku ze starszymi wersjami (jest ich tak wiele że zwyczajnie się nam nie chce nawet próbować opracowywać łatkę aktualizującą 1.3 do 1.4). O ile „stare” problemy mogą zostać potraktowane po macoszemu, to wprowadzone przez nas będą dość porządnie opisane.

Będę się starał pisać zrozumiale dla osoby nieobeznanej z PHP (każdy kiedyś zaczynał).

6.2.1 Problemy z instalacją

te funkcje charakterystyczne dla php5 (coś z datą + zbyt nowe adodb, public/private/static w plikach klas), prawa do katalogów, szczegółowy opis ręcznej instalacji jaki nabazgrałem na forum tech... wskazanie includes/config.php. Prawa zapisu katalogów i plików (dołożyć kat. counter do instalacji)

Dodanie minerału do gry? Ogólnie wkleić/podlinkować ciekawsze aspekty. Może nawet zaprosić do współpracy ludzi którzy kiedyś coś mądrego o tym silniku napisali (kieedyś widziałem jak ktoś PDFa popełnił o instalacji i pisaniu prostego moda. Pewnie Nailo by dał radę go dorwać).

6.2.2 Resety i rzeczy z nimi związane

linki do artów o obsłudze crontaba na serwerze, z panelu lub ssh. webcron.

Zapewne będziesz chciał zmienić ilość resetów. Niby nic trudnego, ustawić częściej crona. Ale silnik jest projektowany pod pewną optymalną ich ilość i więcej niż 7-10 będzie wywoływało pewne zaburzenia w algorytmach, równowadze rozgrywki itd. Załóżmy że będziesz chciał mieć reset co godzinę. To po kolei:

1. **Reset w ogóle nie działa.** Szanujący się cron będzie miał opcję przysłania Ci treści wynikowej strony na maila. Zakładam że już to wypróbowałeś i nie przychodzą komunikaty w stylu „brak takiej strony", „brak dostępu" czy coś. Zakładam również że wywołujesz plik poprawnie, tzn. z adresem reset.php?step=reset lub reset.php?step=revive. Jeśli masz komunikat „Ciekawe co tu chciałeś zrobić", to usuń z pliku reset.php linijki:

```
if ($_SERVER['REMOTE_ADDR'] != '83.142.73.186')
{
      die("Ciekawe co chciałeś tutaj zrobić?");
}
```

Mogą one wyglądać trochę inaczej, ale zasadniczo chodzi tu o zabezpieczenie przed ręcznym wywoływaniem resetów przez nieuczciwych graczy. Usunięcie tego nie jest zbyt bezpieczne, pod koniec rozdziału spróbujemy napisać jakieś proste zabezpieczenie. Najpierw sprawdź czy reset zadziała po wpisaniu słowa localhost zamiast tego IP (albo IP serwera crona, jeśli wywołujesz resety z serwera innego niż ten gry). Jeśli odpalasz resety ręcznie, to warto tu wpisać swoje IP ;)

2. **Sprawdzanie czy „teraz" jest reset.** W includes/head.php znajdź linijki:

```
// Check if game isn't closed for "reset" reasons - only near every odd hour
(0,2,4,6,8 etc.).
$intTime = time() % 7200;
if ($intTime < 60 || $intTime > 7140)
{
    $arrOpen = $db -> GetRow('SELECT `value` FROM `settings` WHERE
`setting`=\'open\'');
    if ($arrOpen['value'] == 'N' && $player -> rank != 'Admin')
    {
        $arrReason = $db -> GetRow('SELECT `value` FROM `settings` WHERE
`setting`=\'close_reason\'');
        $smarty -> assign ('Error', REASON.'<br />'.$arrReason['value']);
        $smarty -> display ('error.tpl');
        exit;
    }
}
```

Jak widać, sprawdzenie „czy przypadkiem nie wykonuje się reset" ma miejsce tylko gdy mamy pełną parzystą godzinę +/- 1 minuta. Pożyteczne to o tyle, że szkoda to zapytanie wykonywać co chwilę, na każdej stronie na jaką trafi gracz. Masz teraz kilka opcji:

- wywalić w całości warunek czasowy, doprowadzając do postaci takiej jak w wersjach < 1.4 (usuń linijki pogrubione),
- zmienić liczbę 7200 na 3600, żeby sprawdzało co godzinę (wtedy trzeba też te 7140 zmienić na 3540 sekund = 59 minut),
- jeśli masz problemy z cronem i potrzebujesz większego marginesu błędu – ustawić więcej niż 60 sekund przed i po pełnej godzinie (czy ten serwer Vallheru dalej dostaje zajoba i się spóźnia?)

3. Nad listą graczy widać **tekst „do resetu pozostało ...”**. Sposób obliczania tej wartości zmienisz w pliku includes/counttime.php. Znajdź w nim zmienną `$arrTime2` i wstaw do listy swoje godziny (wstawiaj rosnąco, oddzielaj przecinkami). Zapisz plik. Od tej chwili zarówno ten tekst, jak i wieża zegarowa (tower.php) i pustelnik wskrzeszający w górach/lesie powinni „działać” poprawnie.

4. Na Arenę Walk (do pliku battle.php) wstawiono **ograniczenie nie pozwalające na atak na gracza tuż po resecie**. Ma to dać każdemu czas na schowanie się w domku, zażycie antidotum, zatrucie broni czy nawet normalną grę (jeśli ktoś gra rzemieślnikiem bez immunitetu i dotychczas był regularnie „ciepany” dwie sekundy po resecie). Znajdź poniższy fragment, znowu masz kilka opcji (coś biednie ten mój kod wygląda. Poprawić? includnąć counttime.php?):

```
// Protection from attacks immediately after reset.
// Will work only with resets at full hour, need to be adapted otherwise.
    $arrResets = array( 0,8,10,12,14,16,18,20,22,24);
    $intTimestamp = time();  // current time (Unix timestamp)
    $year = date("Y");
    $month = date("m");
    $day = date("d");
    $hour = date("H");
    $blnTest = true;
    for($i = 0; $i < 8; $i++) // count to "number of resets" -1 !
        if( $arrResets[$i] <= $hour && $hour < $arrResets[$i+1]) // find between
which resets we are
        {// reset gap was found
            $start = mktime ($arrResets[$i],0,0,$month,$day,$year);
            $stop = mktime ($arrResets[$i],1,0,$month,$day,$year);
            if( $intTimestamp >= $start && $intTimestamp < $stop)
                error(TOO_SOON.' '.$arrResets[$i].':'.'1.');
            break;
        }
```

- powrót do wersji silnika poniżej 1.4 i usunięcie tego (usuń cały ten fragment i linijkę ze słowem TOO_SOON z languages/pl/battle.php),
- dołożenie własnych godzin resetów – oprócz listy jak poprzednio musisz wpisać ich ilość minus

1 w miejsce pogrubionej ósemki,

- zmiana okresu ochronnego – inną liczbę minut wpisz w miejsce pogrubionych jedynek, w pozostałych przypadkach musisz poczytać dokumentację do funkcji mktime.

5. „**Mamy dzień X ery Y**” - można to zmienić w bazie danych, wykonując polecenia SQL (najprościej w phpMyAdmin, ewentualnie konsola sql albo w ostateczności zmodyfikuj jakiś plik gry wstawiając tam odpowiednie `$db -> Execute();` tylko pamiętaj o usunięciu potem):

```
UPDATE `settings` SET `value`=10 WHERE `setting`=`day`; -- dzień dziesiąty
UPDATE `settings` SET `value`=7 WHERE `setting`=`age`;  -- ery siódmej
```

6. Początkowe **ceny** i mechanizm ich naliczania **w Magazynie Królewskim**:

Do wersji 1.3 włącznie MK rozpoczynał erę pusty, a początkowe ceny były wpisane do tabelki `settings` przy odpowiednich nazwach minerałów. Gdy mijał dzień pierwszy zaczęła się wypełniać tabelka `warehouse` zawierająca informacje o cenie, dostępnej ilości i wartościach kupna/sprzedaży minerałów i ziół do 10 dni wstecz. Reset = 1 w tej tabeli oznacza dzień dzisiejszy, a dziesiąty to najstarszy. Podczas resetu głównego (wywołanie pliku reset.php wykonujące kod z includes/resets.php) resety w tabelce `warehouse` są postarzane o 1 i naliczane są nowe ceny. Ceny te są wpisywane zarówno do `settings`, jak i do `warehouse` do nowego pierwszego resetu.

W wersjach tych możesz modyfikować ceny początkowe zmieniając pola w tabelce `settings` wykonując zapytanie podobne do tego z poprzedniego punktu.

Od wersji 1.4 w górę zmieniło się przeznaczenie pól w tabelce `settings`. Zamiast cen (co przecież jest dublowaniem resetu pierwszego) zawierają one licznik dni, w których w MK nie było danego surowca. Dzięki temu licznikowi cena rośnie szybciej. Pojawił się za to problem ustalenia pierwotnych cen i ilości surowców. Znajdź w plikach warehouse.php i includes/resets.php zmienne `$arrSQLNames, $arrStartAmount, $arrStartPrice`. Ich przeznaczenie jest dość oczywiste ;) Najprościej Ci będzie pracować na tych z warehouse.php (najpierw jednak ustaw linijkę `$smarty->caching=0;`, żeby widzieć zmiany na bieżąco; szczegóły w rozdziale o cache Smarty'ego), a jak będziesz zadowolony to skopiuj te zmienne do includes/resets.php.

Niezależnie od wersji algorytm obliczania cen jest ten sam (no, w 1.4+ jeszcze ten „licznik zerowych dni”). Zatem możesz być niezadowolony z ceny jaka się ustali po resecie. Receptą na to jest jakby zaczęcie ery od nowa, tzn. wykonanie polecenia SQL „`TRUNCATE TABLE `warehouse``”. Stare wersje silnika muszą teraz przestawić ceny (a nowe – wyzerować liczniki) w tabelce

`settings`. Zmiana samego algorytmu obliczania jest bardziej skomplikowana, bierz się za to jak nabierzesz wprawy z PHP (w końcu to miała porada dla „świeżych" adminów być ;))

7. Ostatnią rzeczą jaka mi przychodzi do głowy w związku z resetami jest szybkość dojrzewania ziół na farmie. W pliku farm.php odpowiada za to linijka

```
$arrAge = array(0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 10, 10,
10, 9, 8, 8, 7, 7, 6, 6, 5, 5, 4, 4, 3, 3, 2, 2, 1, 1);
```

Im wyższa liczba, tym więcej zbierze ogrodnik, optimum wzrostu wypada w środku tabelki. Zebrane natychmiast dadzą zero zysku, po kolejnych resetach zwykłych – tak jak wynika z tabelki. Skoro chcesz więcej resetów, to pewnie będziesz potrzebował dłuższego czasu dojrzewania. Dodatkowo od wersji 1.4 różne zioła rosną lepiej w różnych miastach, więc jeszcze to można zmienić. Jeśli wydłużyłeś tabelkę, to w includes/resets.php będziesz chciał zmienić maksymalny wiek roślin po którym one więdną:

```
    $db -> Execute("DELETE FROM `farm` WHERE `age`>38");
```

Sprawdzić czy w moich resetach przepisanych nie zostało stare 26! Bo coś czuję ze pracowałem na starej wersji.

8. Na koniec obiecane bieda-zabezpieczenie przed wywoływaniem resetów przez graczy. Najłatwiej jest zmienić nazwę pliku resetów, np. „tesciowa.php" i odpowiednio przerobić wpisy w cronie. Dodatkowo nałożymy sobie hasło na plik. Tzn. oprócz step=reset czy step=revive trzeba podać jeszcze jakąś zmienną, bo inaczej się nie wykona. Warto jednak zaznaczyć że zabezpieczenie po IP jest najlepsze i naprawdę, użyj tego co tu napisałem dopiero gdy wyczerpiesz inne możliwości. „Security by obscurity" to kiepski pomysł.

```
if (!isset($_GET['haslo']) || $_GET['haslo'] != 'abc')
{
 die("Ciekawe co chciałeś tutaj zrobić?");
}

require_once("includes/config.php");
require_once('includes/resets.php');
if ($_GET['step'] == 'reset')
{
 mainreset();     // odpalany przez tesciowa.php?step=reset&haslo=abc
 exit;            // oczywiscie nazwa pliku i „hasło" to Twoja rzecz
}
if ($_GET['step'] == 'revive')
{
 smallreset();
 exit;
}
```

6.3 Lista plików do usunięcia

Tabela 2: Pliki - śmieci, do wywalenia z serwera i Subversion.

Nazwa/lokalizacja	Komentarz
referrals.php	zmienione znaczenie Vallarów
includes/sessions.php	trochę przegięcie cały plik dla jednej instrukcji. Więcej dysk zamieli go szukając i inkludując niż to „uwspólnienie kodu" warte. Znaleźć miejsca występowania i pozastępować.
help.php	Nieaktualne + będzie FAQ i/lub Wiki
kodzior.php (source.php)	Unikamy pokazywania kodu
languages/pl/quest_class.php	zerowy rozmiar
languages/pl/verifipass.php	zerowy rozmiar (pewnie literówka w nazwie)
katalog includes/js	inny katalog dla JavaScriptów
cache/Fuck.html katalogi lang, tpl	Padlismy ofiarą jakichś hakierów? Jesteśmy serwerem spamu?
katalogi templates/layout1, templates_c/layout1	Bezużyteczne (ale wypadałoby stworzyć css imitujący wystrój graficzny Vall, bo Solostran będzie płakał ;))
images/oaza.jpg	mój profil :)
wszystkie (dla linuksiarzy ukryte) katalogi .svn z kat. adodb	Thindil pewnie brał najnowszą wersję ADODB z ich Subversion. Nam niepotrzebne wersjonowanie tego. Marnuje miejsce na serwerze i rozpycha zipy z silnikiem (jak to powywalałem to katalog schudł z 2.35 do 1.16 MB)

Tabela 3: Pliki/katalogi do wyczyszczenia/zmodyfikowania przed wydaniem silnika i/lub startem normalnej gry.

Nazwa/lokalizacja	Komentarz
katalog avatars katalog images/tribes	wiadomo dlaczego (kiedy wreszcie do kat. beasts wstawimy obrazki potworów inne niż Ciasteczkowy? Chociaż kilka, nie wierzę że na necie nie ma żadnego darmowego Szczura) **I który w koncu jest poprawny? avatars/beasts czy images/beasts?**
katalogi cache i templates_c	wiadomo, tymczasówki
katalog counter	Wyzerować wpisy, dopisać go do testu uprawnień podczas instalowania

Nazwa/lokalizacja	Komentarz
katalog images	Obrazków z Wesnoth (weterani w Strażce) nie posiadamy na własność, mogą zostać. Decyzja co do pozostałych należy do autorów (Aranwe, wiórka, trochę moja panna, kto tam jeszcze rysował? Gloin?), ale powinniśmy ich nie usuwać całkowicie tylko zastępować jakimś badziewiem przykładowym. Smileye „blue” są freeware. Widziałem je na paru forach, a zipa z nimi (ściślej: rpm'a) dorwałem na jakiejś stronce poświęconej emotom do któregoś klienta Jabbera dla Mandriva'y. Zresztą w swoim Psi też je mam.
katalog install	zastanowić się nad tekstem udostępnienia? zmienić plik TODO? Dopisać pochodzenie użytych bibliotek JS do thirdparties.txt do skryptu instalacji dopisać test uprawnien do katalogu „counter”
katalog quests	zostawić jakiś poradnik jak pisać (a najlepiej parę przykładowych, obficie skomentowanych)
katalog languages	Proponuję wyczyścić/pozastępować byle czym tylko istotne langi, jak opisy lokacji, teksty karczmarza, treść questów. Możemy usunąć all pliki ale wtedy nikomu się nie będzie chciało tego używać,a bugtracki oszaleją...
Changelog	coś wpisać :D choćby „za dużo żeby wszystko wymienić”
mojelewejajo.php	zmienić nazwę na reset.php. Na serwerze gry: włączyć blokadę IP
install/resetall.php	Wywalić metodę przepisywania Vallarów tylko jeśli gracz wciąż gra. Dodać do pętli „insert into minerals” (Dellas obiecywał że każdy na start coś dostanie). I w takim razie do aktywacja.php też.
zrzuty baz danych	tu trzeba siąść i spokojnie niemal każdą tabelkę przejrzeć. Potwory, questy, nazwy broni itp. Ja sobie tu tylko wpiszę że zrzut musi zawierać inserty do tabelki settings z wyzerowanymi minerałami i ziołami.

6.4 Autorzy

Jeśli nie zaznaczono inaczej, autorem tekstu jest członek Orodlin Team o pseudonimie eyescream (mail: tduda@users.sourceforge.net, Jabber: eyescream@jabber.wroc.pl). Jeśli dany fragment oparty jest na innym opracowaniu (dokumentacja, artykuł, przykładowy kod itp.), jest to zaznaczone.