Wdrażanie potoku zamówień: część 1

Wdrożenie lejka zamówień to pierwszy krok do stworzenia profesjonalnego systemu zarządzania zamówieniami. W tej i następnej Części zbudujemy własny proces przetwarzania zamówień, który zajmuje się autoryzacją kart kredytowych, sprawdzaniem zapasów, wysyłką, powiadamianiem e-mailem i tak dalej. Pokażemy, gdzie ten proces pasuje do obrazu. Funkcjonalność potoku zamówień to niezwykle przydatna funkcja dla witryny e-commerce. Funkcje potoku zamówień pozwalają nam śledzić zamówienia na każdym etapie procesu i dostarczać informacje audytowe, do których możemy się później odwołać lub jeśli coś pójdzie nie tak podczas realizacji zamówienia. Możemy to wszystko zrobić bez polegania na zewnętrznym systemie księgowym, co również może obniżyć koszty. Większość tej Częsci dotyczy tego, czym jest system potoków i konstruowania tego systemu, co wiąże się również z niewielkimi modyfikacjami sposobu, w jaki obecnie działają, oraz kilkoma dodatkami do bazy danych, z której korzystamy. Jednak kod nie jest dużo bardziej skomplikowany niż kod, którego już używaliśmy. Prawdziwym wyzwaniem jest projektowanie systemu. Po zaprojektowaniu potoku zamówień, funkcje, które dodasz do niego to:

• Aktualizacja statusu zamówienia

• Ustawianie szczegółów uwierzytelnienia karty kredytowej

• Ustawienie daty wysyłki zamówienia

• Wysyłanie e-maili do klientów i dostawców

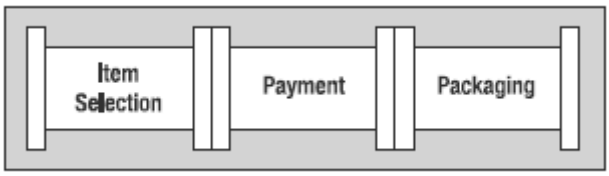
• Pobieranie szczegółów zamówienia i adresu klienta

Pod koniec następnej Części klienci będą mogli składać zamówienia w naszym lejku, a my będziemy mogli śledzić postęp tych zamówień na różnych etapach. Chociaż nie nastąpi jeszcze żadne prawdziwe przetwarzanie kart kredytowych, otrzymamy całkiem kompletny system, w tym nową administracyjną stronę internetową, z której dostawcy mogą korzystać w celu potwierdzenia, że mają produkty w magazynie i potwierdzenia, że zamówienia zostały wysłane . Na początek jednak potrzebujemy nieco więcej informacji na temat tego, co tak naprawdę staramy się osiągnąć.

Co to jest potok zamówień?

Każda transakcja handlowa, czy to w sklepie na ulicy, przez Internet, czy gdziekolwiek indziej, wiąże się z kilkoma powiązanymi zadaniami, które należy wykonać, zanim można ją uznać za zakończoną. Na przykład nie możemy po prostu usunąć ubrania z butiku mody (bez płacenia) i powiedzieć, że go kupiliśmy – wynagrodzenie jest integralną częścią każdego zakupu. Ponadto transakcja zakończy się pomyślnie tylko wtedy, gdy każde z wykonanych zadań zakończy się pomyślnie. Jeśli na przykład karta kredytowa klienta zostanie odrzucona, nie można jej obciążyć żadnymi środkami, więc nie można dokonać zakupu.

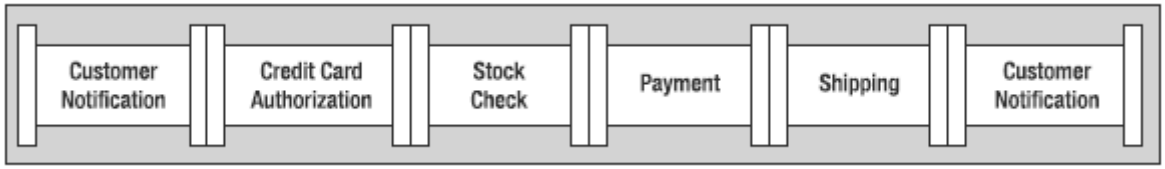
Sekwencja zadań w transakcji jest często rozumiana w kategoriach potoku. W tej analogii zamówienia zaczynają się na jednym końcu rury i wychodzą z drugiego końca po ich zakończeniu. Po drodze muszą przejść przez kilka odcinków potoku, z których każdy odpowiada za określone zadanie lub powiązaną grupę zadań. Jeśli którykolwiek odcinek potoku nie zostanie ukończony, zamówienie „utknie” i może wymagać interakcji z zewnątrz, zanim będzie mógł przejść dalej wzdłuż rurociągu, lub może zostać całkowicie anulowane. Na przykład prosty potok przedstawiony na rysunku dotyczy transakcji w sklepie stacjonarnym.



Ostatnia sekcja, opakowanie, może być opcjonalna i może wiązać się z dodatkowymi zadaniami, takimi jak pakowanie prezentów. Etap płatności może również obejmować jedną z kilku metod działania, ponieważ klient może zapłacić gotówką, kartą kredytową, bony upominkowe i tak dalej. Gdy weźmiemy pod uwagę zakupy w e-commerce, lejek staje się dłuższy, ale tak naprawdę nie jest to bardziej skomplikowane.

Projektowanie potoku zamówień

W aplikacji e-commerce TShirtShop potok będzie wyglądał jak na rysunku.



Zadania realizowane na tych odcinkach potoku są następujące:

Powiadomienie klienta: Do klienta wysyłane jest powiadomienie e-mail z informacją, że rozpoczęło się przetwarzanie zamówienia oraz potwierdzeniem pozycji do wysłania oraz adresem, na który towar zostanie wysłany.

Autoryzacja karty kredytowej: Karta kredytowa użyta do zakupu jest sprawdzana, a całkowita kwota zamówienia jest odkładana na bok (chociaż na tym etapie nie jest pobierana żadna płatność).

Kontrola stanów magazynowych: Do dostawcy wysyłana jest wiadomość e-mail z listą zamówionych artykułów. Przetwarzanie jest kontynuowane, gdy dostawca potwierdzi, że towary są dostępne.

Płatność: Transakcja kartą kredytową jest realizowana z wykorzystaniem wcześniej odłożonych środków.

Wysyłka: Do dostawcy wysyłana jest wiadomość e-mail z potwierdzeniem dokonania płatności za zamówione artykuły. Przetwarzanie jest kontynuowane, gdy dostawca potwierdzi, że towary zostały wysłane.

Powiadomienie klienta: Wysyłana jest wiadomość e-mail z powiadomieniem klienta o wysłaniu zamówienia i podziękowaniem za korzystanie ze strony internetowej TShirtShop.

Uwaga : Jeśli chodzi o wdrożenie, jak wkrótce zobaczysz, jest więcej etapów, ponieważ etapy kontroli zapasów i wysyłki w rzeczywistości składają się z dwóch odcinków rurociągu – jednej, która wysyła wiadomość e-mail, a drugiej, która czeka na potwierdzenie.

Gdy zamówienia przepływają przez ten potok, wpisy są dodawane do nowej tabeli bazy danych o nazwie audyt. Te wpisy można sprawdzić, aby zobaczyć, co się stało z zamówieniem i są doskonałym sposobem na zidentyfikowanie problemów, jeśli wystąpią. Każdy wpis w tabeli zamówień jest również oflagowany statusem, identyfikującym punkt w potoku, do którego dotarł. Aby przetworzyć potok, stworzymy klasy reprezentujące każdy etap. Klasy te przeprowadzają wymagane przetwarzanie, a

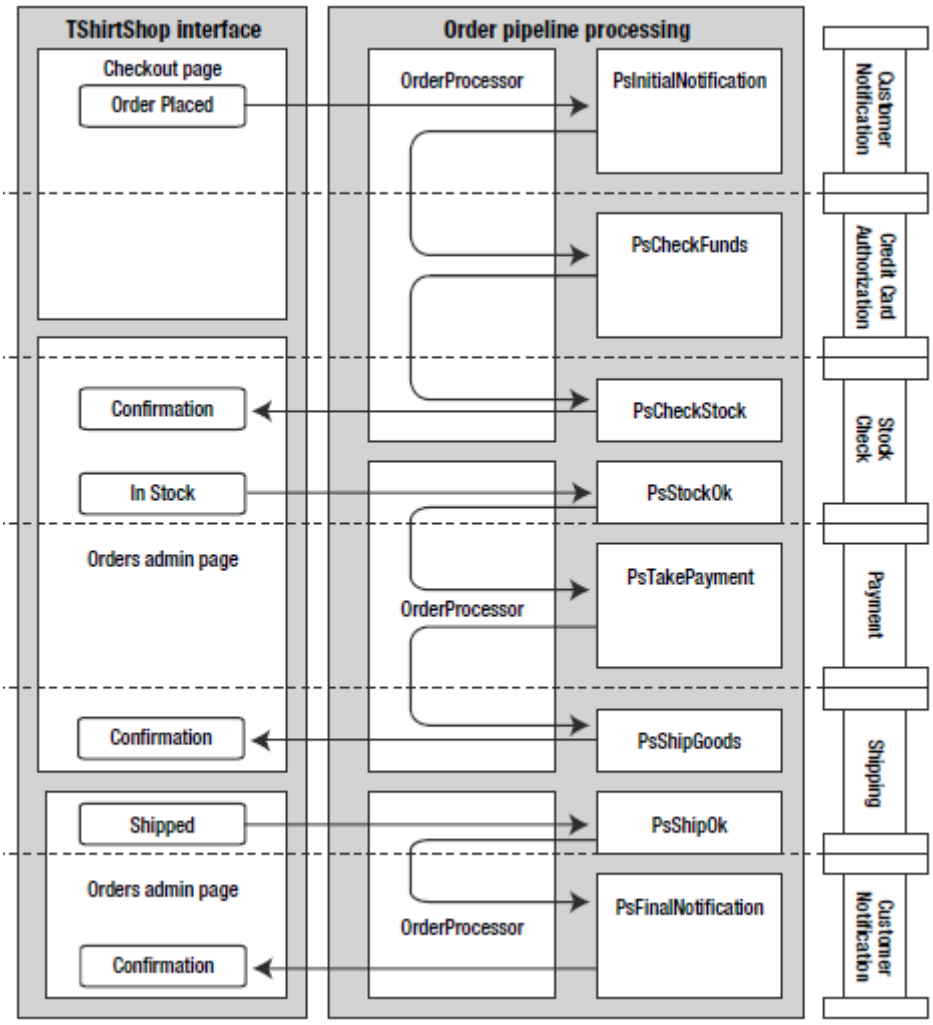
następnie modyfikują status zamówienia w tabeli zamówień, aby przesunąć zamówienie. Będziemy również potrzebować klasy koordynującej (lub procesora), która może zostać wywołana dla dowolnego zamówienia i wykona odpowiednią klasę etapu potoku. Procesor ten jest wywoływany raz przy składaniu zamówienia i podczas normalnej pracy jest wywoływany dwa razy - raz w celu potwierdzenia stanu magazynowego i raz w celu potwierdzenia wysyłki. Aby ułatwić życie, zdefiniujemy również wspólny interfejs obsługiwany przez każdą klasę etapu potoku. Umożliwia to klasie procesora zamówień dostęp do każdego etapu w standardowy sposób. Zdefiniujemy również kilka funkcji narzędziowych i ujawnimy kilka wspólnych właściwości w klasie procesora zamówień, które będą używane w razie potrzeby przez etapy potoku. Na przykład identyfikator zamówienia powinien być dostępny dla wszystkich etapów potoku, więc aby uniknąć duplikacji kodu, umieścimy tę informację w klasie procesora zamówień. Przejdźmy teraz do konkretów. Zbudujemy wiele plików w folderze biznesowym zawierającym wszystkie nowe klasy, do których będziemy się odwoływać z TShirtShop. Nowe pliki, które utworzymy, to:

OrderProcessor: Główna klasa do przetwarzania zamówień.

IPipelineSection: definicja interfejsu dla odcinków potoku.

PsInitialNotification, PsCheckFunds, PsCheckStock, PsStockOk, PsTakePayment, PsShipGoods, PsShipOk, PsFinalNotification: Klasy sekcji potoku. Stworzymy te klasy w rozdziale 19; tutaj zamiast tego użyjemy klasy dummy (PsDummy).

Postęp zamówienia przez potok, za pośrednictwem procesora zamówienia, odnosi się do potoku pokazanego wcześniej.



Proces przedstawiony na tym schemacie podzielony jest na trzy sekcje:

• Klient składa zamówienie.

• Dostawca potwierdza stan magazynowy.

• Dostawca potwierdza wysyłkę.

Pierwszy etap wygląda następująco:

1. Gdy klient potwierdzi zamówienie, Presentation/checkout_info.php tworzy zamówienie w bazie danych i wywołuje OrderProcessor w celu rozpoczęcia przetwarzania zamówienia.

2. OrderProcessor wykrywa, że zamówienie jest nowe i wywołuje PsInitialNotification.

3. PsInitialNotification wysyła do klienta wiadomość e-mail z potwierdzeniem złożenia zamówienia i przyśpiesza etap realizacji zamówienia. Instruuje również OrderProcessor, aby kontynuował przetwarzanie.

4. OrderProcessor wykrywa status nowego zamówienia i wywołuje PsCheckFunds.

5. PsCheckFunds sprawdza, czy środki są dostępne na karcie kredytowej klienta i przechowuje dane wymagane do sfinalizowania transakcji, jeśli środki są dostępne. Jeśli to się powiedzie, etap

zamówienia jest zaawansowany, a OrderProcessor otrzymuje polecenie kontynuowania. Karta kredytowa klienta nie jest jeszcze obciążana.

6. OrderProcessor wykrywa status nowego zamówienia i wywołuje PsCheckStock.

7. PsCheckStock wysyła wiadomość e-mail do dostawcy z listą zamówionych pozycji, instruuje dostawcę, aby potwierdził za pośrednictwem ADMINISTRACJI ZAMÓWIEŃ z sekcji administratora i przesuwa status zamówienia.

8. Procesor Zamówień zostaje rozwiązany.

Drugi etap wygląda następująco:

1. Gdy dostawca zaloguje się na stronie administratora zamówień, aby potwierdzić dostępność towaru, Presentation/admin_order_details.php wywołuje OrderProcessor, aby kontynuować przetwarzanie zamówienia.

2. Jeśli dostawca potwierdzi dostępność towaru, OrderProcessor wykrywa nowy status zamówienia i wywołuje PsStockOk.

3. PsStockOk przesuwa status zamówienia i informuje OrderProcessor, aby kontynuował.

4. OrderProcessor wykrywa status nowego zamówienia i wywołuje PsTakePayment.

5. PsTakePayment wykorzystuje szczegóły transakcji zapisane wcześniej przez PsCheckFunds, aby zakończyć transakcję, obciążając kartę kredytową klienta za zamówienie, a następnie przesuwa status zamówienia, informując OrderProcessor, aby kontynuował.

6. OrderProcessor wykrywa status nowego zamówienia i wywołuje PsShipGoods.

7. PsShipGoods wysyła do dostawcy wiadomość e-mail z potwierdzeniem zamówionych pozycji, poleca dostawcy wysyłkę tych towarów do klienta oraz informuje o statusie zamówienia.

8. Procesor Zamówień zostaje rozwiązany.

Trzeci etap wygląda następująco:

1. Gdy dostawca potwierdzi, że towar został wysłany, Presentation/ admin_order_details.php wywołuje OrderProcessor, aby kontynuować przetwarzanie zamówienia.

2. OrderProcessor wykrywa status nowego zamówienia i wywołuje PsShipOk.

3. PsShipOk wprowadza datę wysyłki do bazy danych, przesuwa status zamówienia i mówi OrderProcessorowi, aby kontynuował.

4. OrderProcessor wykrywa status nowego zamówienia i wywołuje PsFinalNotification.

5. PsFinalNotification wysyła do klienta wiadomość e-mail z potwierdzeniem wysyłki zamówienia i przyśpiesza etap realizacji zamówienia.

6. Procesor Zamówień zostaje rozwiązany.

Jeśli coś pójdzie nie tak w dowolnym momencie przetwarzania, na przykład odrzucenie karty kredytowej, do administratora wysyłana jest wiadomość e-mail. Administrator ma wówczas wszystkie niezbędne informacje, aby sprawdzić, co się stało, skontaktować się z zaangażowanym klientem i w razie potrzeby anulować lub wymienić zamówienie. Żaden punkt w tym procesie nie jest szczególnie skomplikowany; po prostu potrzeba dużo kodu, aby wprowadzić to w życie!

Układanie podbudowy

Zanim zaczniemy budować opisane komponenty, musimy dokonać kilku modyfikacji w bazie danych i aplikacji internetowej TShirtShop. Podczas przetwarzania zamówień jedną z najważniejszych funkcji potoku jest utrzymywanie aktualnej ścieżki audytu. Implementacja tej ścieżki audytu obejmuje dodawanie rekordów do nowej tabeli bazy danych o nazwie audyt. W poniższym ćwiczeniu dodamy tabelę audytu. Aby zaimplementować opisaną funkcjonalność, musimy również dodać nową funkcję o nazwie orders_create_audit do bazy danych tshirtshop. Procedura składowana orders_create_audit dodaje wpis do tabeli inspekcji. Stworzymy również klasę OrderProcessor (klasę odpowiedzialną za przenoszenie zamówienia przez potok), która zawiera dużo kodu. Możemy jednak zacząć po prostu i w razie potrzeby rozbudowywać dodatkowe funkcje. Na początek stworzymy wersję klasy OrderProcessor z następującą funkcjonalnością:

• Dynamicznie wybiera odcinek potoku obsługujący interfejs IPipelineSection

• Dodaje podstawowe dane audytowe

• Daje dostęp do aktualnych szczegółów zamówienia

• Daje dostęp do klienta dla bieżącego zamówienia

• Daje dostęp do mailingu administratora

• Wysyła wiadomość e-mail do administratora w przypadku błędu

INTERFEJSY W PHP

Po raz pierwszy pracujemy z interfejsami. Interfejsy stanowią wspólną cechę współczesnych języków obiektowych. Interfejs reprezentuje zestaw metod, które klasa musi zdefiniować podczas implementacji interfejsu. Kiedy klasa implementuje interfejs, wymagane jest zaimplementowanie wszystkich metod zdefiniowanych przez ten interfejs. W ten sposób interfejs staje się kontraktem, który gwarantuje, że klasy, które go implementują, zawierają określony zestaw metod. Na przykład w poniższym ćwiczeniu utworzymy interfejs o nazwie IPipelineSection, który zawiera pojedynczą metodę o nazwie Process():

```
interface IPipelineSection

{

public function Process($processor);

}
```

Zaimplementujemy ten interfejs we wszystkich klasach reprezentujących sekcje potoku, upewniając się, że każda z tych klas będzie zawierała metodę o nazwie Process(). W ten sposób, pracując z klasami potoku zamówień, będziemy mogli bezpiecznie wywołać na nich metodę Process(), ponieważ będziemy mieć gwarancję, że ta metoda tam będzie. Interfejs nie może być tworzony jak normalna klasa, ponieważ nie zawiera żadnych implementacji metod, a jedynie ich sygnatury. (Podpis metody to po prostu definicja metody bez kodu). Klasy implementują interfejsy za pomocą operatora implements. Klasa może implementować wiele interfejsów, ale te interfejsy nie mogą zawierać tej samej metody, aby uniknąć niejednoznaczności. Stworzymy pojedynczy odcinek potoku, PsDummy, który wykorzystuje część tej funkcjonalności. PsDummy jest używany w kodzie zamiast prawdziwych klas sekcji potoku, które zaimplementujemy w następnej Części.

Ćwiczenie: Implementacja szkieletu funkcjonalności przetwarzania zleceń

1. Używając phpMyAdmin, wybierz bazę danych tshirtshop i otwórz nową stronę zapytań SQL.

2. Wykonaj ten kod, który utworzy tabelę audytu w bazie danych tshirtshop:

```
-- Create audit table

CREATE TABLE `audit` (

`audit_id` INT NOT NULL AUTO_INCREMENT,

`order_id` INT NOT NULL,

`created_on` DATETIME NOT NULL,

`message` TEXT NOT NULL,

`code` INT NOT NULL,

PRIMARY KEY (`audit_id`),

KEY `idx_audit_order_id` (`order_id`)

);
```

3. Wykonaj następujący kod, który utworzy procedurę składowaną orders_create_audit w bazie danych tshirtshop (nie zapomnij ustawić ogranicznika na $$):

```
-- Create orders_create_audit stored procedure

CREATE PROCEDURE orders_create_audit(IN inOrderId INT,

IN inMessage TEXT, IN inCode INT)

BEGIN

INSERT INTO audit (order_id, created_on, message, code)

VALUES (inOrderId, NOW(), inMessage, inCode);

END$$
```

4. Przechodząc do warstwy biznesowej, dodaj następującą metodę do klasy Orders w business/orders.php:

```
// Creates audit record

public static function CreateAudit($orderId, $message, $code)

{

// Build the SQL query

$sql = 'CALL orders_create_audit(:order_id, :message, :code)';

// Build the parameters array

$params = array (':order_id' => $orderId,

':message' => $message,

':code' => $code);
```

```
// Execute the query
DatabaseHandler::Execute($sql, $params);
}
```

5. Dodaj nowy plik do katalogu biznesowego o nazwie order_processor.php z następującym kodem:

```
<?php
/* Main class, used to obtain order information,
run pipeline sections, audit orders, etc. */
class OrderProcessor
{
public $mOrderInfo;
public $mOrderDetailsInfo;
public $mCustomerInfo;
public $mContinueNow;
private $_mCurrentPipelineSection;
private $_mOrderProcessStage;
// Class constructor
public function __construct($orderId)
{
// Get order
$this->mOrderInfo = Orders::GetOrderInfo($orderId);
if (empty ($this->mOrderInfo['shipping_id']))
$this->mOrderInfo['shipping_id'] = -1;
if (empty ($this->mOrderInfo['tax_id']))
$this->mOrderInfo['tax_id'] = -1;
// Get order details
$this->mOrderDetailsInfo = Orders::GetOrderDetails($orderId);
// Get customer associated with the processed order
$this->mCustomerInfo = Customer::Get($this->mOrderInfo['customer_id']);
$credit_card = new SecureCard();
$credit_card->LoadEncryptedDataAndDecrypt(
$this->mCustomerInfo['credit_card']);
```

```
$this->mCustomerInfo['credit_card'] = $credit_card;
}
/* Process is called from presentation/checkout_info.php and
presentation/admin_orders.php to process an order */
public function Process()
{
// Configure processor
$this->mContinueNow = true;
// Log start of execution
$this->CreateAudit('Order Processor started.', 10000);
// Process pipeline section
try
{
while ($this->mContinueNow)
{
$this->mContinueNow = false;
$this->_GetCurrentPipelineSection();
$this->_mCurrentPipelineSection->Process($this);
}
}
catch(Exception $e)
{
$this->MailAdmin('Order Processing error occurred.',
'Exception: "' . $e->getMessage() . '" on ' .
$e->getFile() . ' line ' . $e->getLine(),
$this->_mOrderProcessStage);
$this->CreateAudit('Order Processing error occurred.', 10002);
throw new Exception('Error occurred, order aborted. ' .
'Details mailed to administrator.');
}
$this->CreateAudit('Order Processor finished.', 10001);
```

```
}
// Adds audit message
public function CreateAudit($message, $code)
{
Orders::CreateAudit($this->mOrderInfo['order_id'], $message, $code);
}
// Builds e-mail message
public function MailAdmin($subject, $message, $sourceStage)
{
$to = ADMIN_EMAIL;
$headers = 'From: ' . ORDER_PROCESSOR_EMAIL . "\r\n";
$body = 'Message: ' . $message . "\n" .
'Source: ' . $sourceStage . "\n" .
'Order ID: ' . $this->mOrderInfo['order_id'];
$result = mail($to, $subject, $body, $headers);
if ($result === false)
{
throw new Exception ('Failed sending this mail to administrator:' .
"\n" . $body);
}
}
// Gets current pipeline section
private function _GetCurrentPipelineSection()
{
$this->_mOrderProcessStage = 100;
$this->_mCurrentPipelineSection = new PsDummy();
}
}
?>
```

6. Utwórz interfejs IPipelineSection w nowym pliku o nazwie business/i_pipeline_section.php w następujący sposób:

```
<?php
interface IPipelineSection
{
public function Process($processor);
}
?>
```

7. Dodaj nowy plik w katalogu biznesowym o nazwie ps_dummy.php z następującym kodem. Klasa PsDummy jest używana w tym rozdziale do celów testowych zamiast rzeczywistych sekcji potoku, które zaimplementujemy w następnym rozdziale.

```
<?php
class PsDummy implements IPipelineSection
{
public function Process($processor)
{
$processor->CreateAudit('PsDoNothing started.', 99999);
$processor->CreateAudit('Customer: ' .
$processor->mCustomerInfo['name'], 99999);
$processor->CreateAudit('Order subtotal: ' .
$processor->mOrderInfo['total_amount'], 99999);
$processor->MailAdmin('Test.', 'Test mail from PsDummy.', 99999);
$processor->CreateAudit('PsDoNothing finished', 99999);
}
}
?>
```

8. Dodaj następujący kod do pliku include/config.php, dostosowując dane do własnych adresów e-mail:

```
// Constant definitions for order handling related messages
define('ADMIN_EMAIL', 'Admin@example.com');
define('CUSTOMER_SERVICE_EMAIL', 'CustomerService@example.com');
define('ORDER_PROCESSOR_EMAIL', 'OrderProcessor@example.com');
define('SUPPLIER_EMAIL', 'Supplier@example.com');
```

Uwaga : Wartości ADMIN_EMAIL i SUPPLIER_EMAIL będą faktycznie używane do wysyłania wiadomości e-mail. Innymi słowy, muszą to być prawdziwe adresy e-mail, które możesz zweryfikować.

Możesz pozostawić CUSTOMER_SERVICE_EMAIL i ORDER_PROCESSOR_EMAIL bez zmian, ponieważ są one używane w polu FROM wiadomości e-mail i nie muszą to być prawidłowe adresy e-mail.

9. Dodaj podświetlone linie do admin.php

```
// Load Business Tier
...
require_once BUSINESS_DIR . 'customer.php';
require_once BUSINESS_DIR . 'i_pipeline_section.php';
require_once BUSINESS_DIR . 'ps_dummy.php';
require_once BUSINESS_DIR . 'order_processor.php';
```

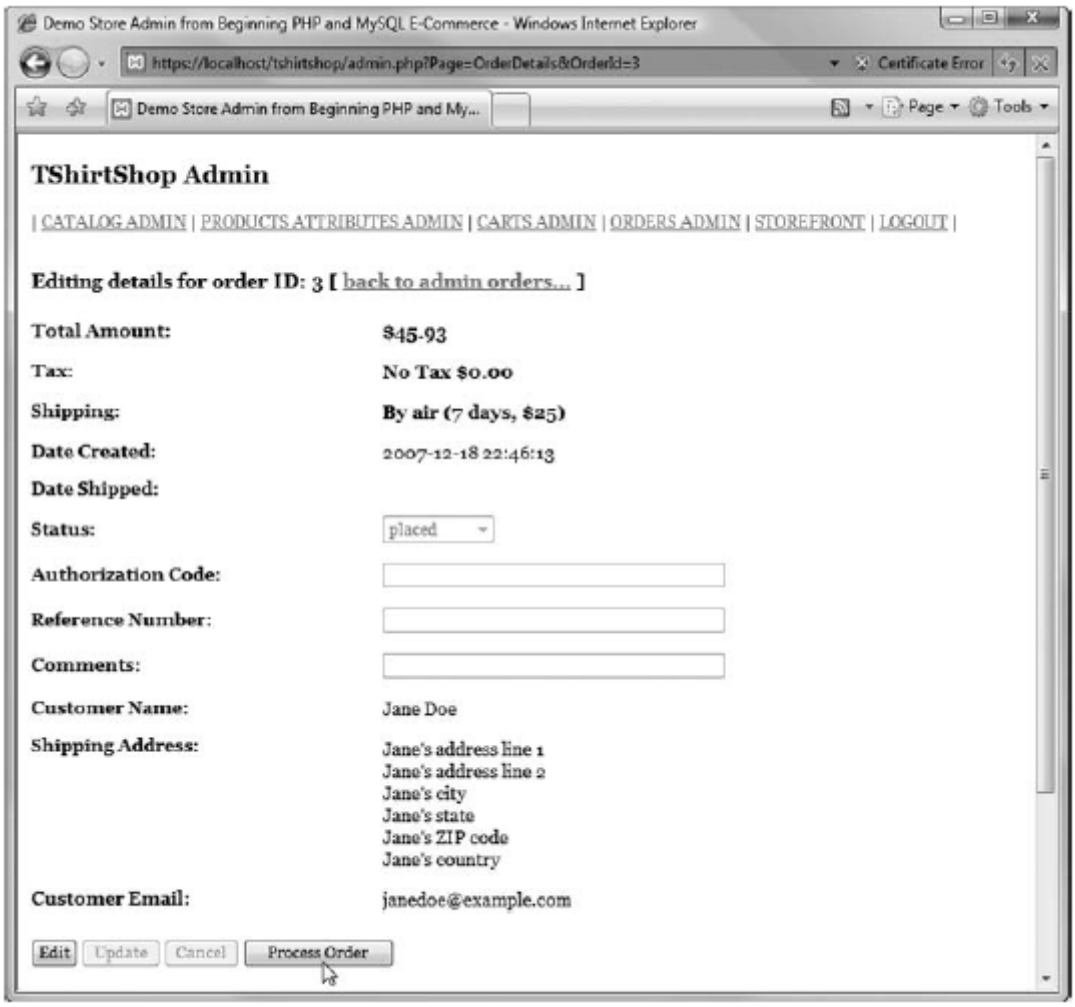
10. Zmodyfikuj prezentację/templates/admin_order_details.tpl dodając podświetloną linię:

```
<input type="submit" name="submitCancel" value="Cancel"
{if ! $obj->mEditEnabled} disabled="disabled" {/if} />
<input type="submit" name="submitProcessOrder" value="Process Order" />
</p>
<h3>Order contains these products:</h3>
```

11. Zmodyfikuj prezentację/admin_order_details.php, jak zaznaczono tutaj:

```
// Initializes class members
public function init()
{
if (isset ($_GET['submitUpdate']))
{
Orders::UpdateOrder($this->mOrderId, $_GET['status'],
$_GET['comments'], $_GET['authCode'], $_GET['reference']);
}
if (isset ($_GET['submitProcessOrder']))
{
$processor = new OrderProcessor($this->mOrderId);
$processor->Process();
}
$this->mOrderInfo = Orders::GetOrderInfo($this->mOrderId);
$this->mOrderDetails = Orders::GetOrderDetails($this->mOrderId);
```

12. Załaduj stronę administrowania zamówieniami w przeglądarce i wybierz zamówienie, aby wyświetlić jego szczegóły. Na stronie szczegółów zamówienia kliknij przycisk Przetwarzaj zamówienie.



Uwaga : Jeśli chcesz mieć możliwość wysłania wiadomości e-mail z błędem na konto pocztowe hosta lokalnego (twoja_nazwa@locahost), wówczas na komputerze powinien być uruchomiony serwer SMTP (Simple Mail Transfer Protocol). W dystrybucji Red Hat (lub Fedora) Linux możesz uruchomić serwer SMTP za pomocą następującego polecenia: service sendmail start

Uwaga : W systemach Windows należy sprawdzić w Menedżerze Internetowych usług informacyjnych (IIS) domyślny wirtualny serwer SMTP i upewnić się, że jest on uruchomiony.

13. Sprawdź swoją skrzynkę odbiorczą w poszukiwaniu nowego e-maila, który powinien brzmieć „Testuj pocztę od PsDummy”.

14. Sprawdź tabelę kontroli w bazie danych, aby zobaczyć nowe wpisy.

←T→	audit_id	order_id	created_on	message	code
	15	3	2007-12-18 22:47:41	Order Processor started.	10000
	16	3	2007-12-18 22:47:41	PsDoNothing started.	99999
	17	3	2007-12-18 22:47:41	Customer: Jane Doe	99999
	18	3	2007-12-18 22:47:41	Order subtotal: 45.93	99999
	19	3	2007-12-18 22:47:41	PsDoNothing finished	99999
	20	3	2007-12-18 22:47:41	Order Processor finished.	10001

Jak to działa: szkielet funkcjonalności przetwarzania zleceń

Wpisy będą dodawane przez OrderProcessor i poszczególne etapy potoku, aby wskazać sukcesy i niepowodzenia. Wpisy te można następnie sprawdzić, aby zobaczyć, co się stało z zamówieniem, co jest ważną funkcją, jeśli chodzi o sprawdzanie błędów. Kolumna kodu jest interesująca, ponieważ pozwala nam skojarzyć określone wiadomości z numerem identyfikacyjnym. Możemy mieć inną tabelę bazy danych, w której te numery kodów będą pasować do opisów, chociaż nie jest to naprawdę konieczne, ponieważ schemat używany do numerowania (jak patrz dalej w rozdziale) jest dość opisowy. Ponadto mamy kolumnę wiadomości, która już zawiera informacje czytelne dla człowieka. W celach demonstracyjnych ustawiamy adresy e-mail administratora i dostawcy na fikcyjne adresy e-mail, które powinny być również adresem klienta używanym do generowania zamówień testowych. Powinniśmy to zrobić, aby upewnić się, że wszystko działa poprawnie przed wysłaniem poczty do świat zewnętrzny. Przyjrzyjmy się teraz klasie OrderProcessor. Głównym ciałem klasy OrderProcessor jest metoda Process(), która jest teraz wywoływana z Presentation/admin_order_details.php w celu przetworzenia zamówienia:

```
public function Process()

{

// Configure processor

$this->mContinueNow = true;
```

Następnie użyliśmy metody CreateAudit(), aby dodać wpis audytu wskazujący, że OrderProcessor został uruchomiony:

```
// Log start of execution

$this->CreateAudit('Order Processor started.', 10000);
```

Następnie dochodzimy do samego przetwarzania zamówienia. Zastosowany tutaj model polega na sprawdzeniu pola logicznego $mContinueNow przed przetworzeniem sekcji potoku. Dzięki temu sekcje mogą określić, czy przetwarzanie powinno być kontynuowane po zakończeniu bieżącego zadania (ustawiając $mContinueNow na true), czy też przetwarzanie powinno zostać wstrzymane (ustawiając $mContinueNow na false). Jest to konieczne, ponieważ podczas sprawdzania, czy produkty są w magazynie i czy środki są dostępne na karcie kredytowej klienta, musimy czekać na dane wejściowe z zewnątrz w określonych punktach rurociągu. Sekcja potoku do przetworzenia jest wybierana przez prywatną metodę _GetCurrentPipelineSection(), która ostatecznie zwraca klasę sekcji potoku (te klasy zbudujemy w następnym rozdziale) odpowiadającą aktualnemu statusowi zamówienia. Jednak w tym momencie _GetCurrentPipelineSection() ma za zadanie ustawić etap procesu i zwrócić instancję PsDummy. W następnym rozdziale zaimplementujemy klasy reprezentujące każdą sekcję potoku i zwrócimy jedną z tych klas zamiast PsDummy.

```
// Gets current pipeline section
private function _GetCurrentPipelineSection()
{
$this->_mOrderProcessStage = 100;
$this->_mCurrentPipelineSection = new PsDummy();
}
```

Wracając do Process(), widzimy tę metodę wywoływaną w bloku try:

```
// Process pipeline section
try
{
while ($this->mContinueNow)
{
$this->mContinueNow = false;
$this->_GetCurrentPipelineSection();
$this->_mCurrentPipelineSection->Process($this);
}
}
```

Zauważ, że $mContinueNow jest ustawione na false w pętli while - domyślnym zachowaniem jest zatrzymanie po każdej sekcji potoku. Jednak wywołanie metody Process() bieżącej klasy sekcji potoku (która otrzymuje parametr bieżącej instancji OrderProcessor, dzięki czemu ma dostęp do elementu $mContinueNow) zmienia wartość $mContinueNow z powrotem na true w przypadku, gdy przetwarzanie powinno przejdź do następnej sekcji potoku bez czekania na interakcję użytkownika. Zauważ, że w poprzednim fragmencie kodu metoda Process() jest wywoływana bez wiedzy, do jakiego rodzaju obiektu odwołuje się $this->_mCurrentPipelineSection. Każda sekcja potoku jest reprezentowana przez inną klasę, ale wszystkie te klasy muszą udostępniać metodę o nazwie Process(). Kiedy takie zachowanie jest potrzebne, standardową techniką jest utworzenie interfejsu, który definiuje wspólne zachowanie, którego potrzebujemy w tym zestawie klas. Wszystkie klasy sekcji potoku zamówień obsługują prosty interfejs IPipelineSection, zdefiniowany w następujący sposób:

```
<?php
interface IPipelineSection
{
public function Process($processor);
}
?>
```

Wszystkie sekcje potoku używają metody Process() do wykonywania swojej pracy. Ta metoda wymaga odwołania OrderProcessor jako parametru, ponieważ sekcje potoku wymagają dostępu do publicznych pól i metod udostępnianych przez klasę OrderProcessor. Ostatnia część metody Process() w OrderProcessor obejmuje przechwytywanie wyjątków. Tutaj przechwytujemy wszelkie wyjątki, które mogą być zgłoszone przez klasy sekcji potoku zamówień i reagujemy na nie wysyłając e-mail do administratora za pomocą metody MailAdmin(), dodając wpis audytu i wyrzucając nowy wyjątek, który może zostać przechwycony przez strony PHP, które używają klasy OrderProcessor:

```
catch(Exception $e)

{

$this->MailAdmin('Order Processing error occurred.',

'Exception: "' . $e->getMessage() . '" on ' .

$e->getFile() . ' line ' . $e->getLine(),

$this->_mOrderProcessStage);

$this->CreateAudit('Order Processing error occurred.', 10002);

throw new Exception('Error occurred, order aborted. ' .

'Details mailed to administrator.');

}
```

Niezależnie od tego, czy przetwarzanie zakończyło się pomyślnie, dodajemy końcowy wpis audytu mówiący, że przetwarzanie zostało zakończone:

```
$this->CreateAudit('Order Processor finished.', 10001);

}
```

Let’s now look at the MailAdmin() method that simply takes a few parameters for the basic e-mail properties:

```
// Builds e-mail message

public function MailAdmin($subject, $message, $sourceStage)

{

$to = ADMIN_EMAIL;

$headers = 'From: ' . ORDER_PROCESSOR_EMAIL . "\r\n";

$body = 'Message: ' . $message . "\n" .

'Source: ' . $sourceStage . "\n" .

'Order ID: ' . $this->mOrderInfo['order_id'];

$result = mail($to, $subject, $body, $headers);

if ($result === false)

{
```

```
throw new Exception ('Failed sending this mail to administrator:' .

"\n" . $body);

}

}
```

Metoda CreateAudit() jest również prosta i wywołuje pokazaną wcześniej metodę warstwy biznesowej Orders::CreateAudit():

```
// Adds audit message

public function CreateAudit($message, $code)

{

Orders::CreateAudit($this->mOrderInfo['order_id'], $message, $code);

}
```

W tym miejscu warto przyjrzeć się schematowi kodu, który wybraliśmy do audytów realizacji zamówień. We wszystkich przypadkach kod audytu będzie liczbą pięciocyfrową. Pierwsza cyfra tego numeru to 1, jeśli audyt jest dodawany przez OrderProcessor lub 2, jeśli audyt jest dodawany przez sekcję potoku. Kolejne dwie cyfry są używane dla etapu potoku, który dodał audyt (który jest bezpośrednio odwzorowywany na status zamówienia w momencie dodania audytu). Ostatnie dwie cyfry jednoznacznie identyfikują wiadomość w tym zakresie. Na przykład do tej pory widzieliśmy następujące kody:

• 10000: Uruchomiono przetwarzanie zamówień

• 10001: Zakończono przetwarzanie zamówień

• 10002: Wystąpił błąd procesora zamówień

Później zobaczymy wiele z tych kodów, które zaczynają się od 2, gdy przejdziemy do sekcji potoku i dołączymy niezbędne informacje do identyfikacji sekcji potoku, jak wspomniano wcześniej. Mamy nadzieję, że zgodzisz się, że ten schemat pozwala na wiele elastyczności, chociaż oczywiście możemy używać dowolnych kodów, które uznamy za stosowne. Na koniec, kody kończące się na 00 i 01 są używane do rozpoczynania i kończenia wiadomości zarówno dla procesora zamówień, jak i etapów potoku, podczas gdy 02 i wyższe są dla innych wiadomości. Nie ma ku temu żadnego realnego powodu poza spójnością między składnikami. Klasa PsDummy, która jest używana w tym procesorze szkieletowym, wykonuje kilka podstawowych funkcji, aby sprawdzić, czy wszystko działa poprawnie:

```
<?php

class PsDummy implements IPipelineSection

{

public function Process($processor)

{

$processor->CreateAudit('PsDoNothing started.', 99999);

$processor->CreateAudit('Customer: ' .
```

```
$processor->mCustomerInfo['name'], 99999);

$processor->CreateAudit('Order subtotal: ' .

$processor->mOrderInfo['total_amount'], 99999);

$processor->MailAdmin('Test.', 'Test mail from PsDummy.', 99999);

$processor->CreateAudit('PsDoNothing finished', 99999);

}

}

?>
```

Kod tutaj używa metod CreateAudit() i MailAdmin() OrderProcessor, aby wygenerować coś, co pokaże, że kod został wykonany poprawnie. Zauważ, że schematy kodu opisane wcześniej nie są tam używane, ponieważ nie jest to prawdziwa sekcja potoku!

To było dużo kodu do przebrnięcia, ale to sprawiło, że kod klienta był bardzo prosty. Poza ustawieniem wszystkich szczegółów konfiguracji, niewiele jest do zrobienia, ponieważ OrderProcessor wykonuje za Ciebie dużo pracy. Zauważ, że kod, który otrzymaliśmy, jest w większości konsekwencją dokonanych wcześniej wyborów projektowych. Jest to doskonały przykład tego, jak silny projekt może doprowadzić Cię prosto do potężnego i solidnego kodu.

Aktualizacja kodu przetwarzania zamówień

Musimy dodać jeszcze kilka drobiazgów do klasy OrderProcessor, a zrobimy to wykonując kilka krótkich ćwiczeń. Ćwiczenia te wdrażają funkcje wymienione na początku:

• Aktualizacja statusu zamówienia

• Ustawianie szczegółów uwierzytelnienia karty kredytowej

• Ustawienie daty wysyłki zamówienia

• Wysyłanie e-maili do klientów i dostawców

• Pobieranie szczegółów zamówienia i adresu klienta

Zaczniemy od napisania kodu, który umożliwia aktualizację statusu zamówienia. Każdy odcinek rurociągu wymaga możliwości zmiany statusu zamówienia, przesuwając go do następnego odcinka rurociągu. Zamiast po prostu zwiększać status, ta funkcjonalność jest elastyczna, na wypadek, gdybyśmy otrzymali bardziej skomplikowany, rozgałęziony potok. Wymaga to nowej procedury składowanej w bazie danych o nazwie orders_update_status i metody warstwy biznesowej UpdateOrderStatus(), którą musimy dodać do klasy Orders (znajdującej się w business/orders.php).

Ćwiczenie: Aktualizacja statusu zamówienia

1. Zacznij od utworzenia procedury składowanej orders_update_status w bazie danych tshirtshop:

```
-- Create orders_update_status stored procedure

CREATE PROCEDURE orders_update_status(IN inOrderId INT, IN inStatus INT)

BEGIN
```

```
UPDATE orders SET status = inStatus WHERE order_id = inOrderId;
END$$
```

2. Dodaj metodę UpdateOrderStatus() do klasy Orders w business/orders.php:

```
// Updates the order pipeline status of an order
public static function UpdateOrderStatus($orderId, $status)
{
// Build the SQL query
$sql = 'CALL orders_update_status(:order_id, :status)';
// Build the parameters array
$params = array (':order_id' => $orderId, ':status' => $status);
// Execute the query
DatabaseHandler::Execute($sql, $params);
}
```

3. Metoda w OrderProcessor (w business/order_processor.php), która wywołuje tę metodę warstwy biznesowej, jest również nazywana UpdateOrderStatus(). Dodaj tę metodę do order_processor.php:

```
// Set order status
public function UpdateOrderStatus($status)
{
Orders::UpdateOrderStatus($this->mOrderInfo['order_id'], $status);
$this->mOrderInfo['status'] = $status;
}
```

Ćwiczenie: Ustawianie szczegółów uwierzytelniania karty kredytowej

1. Najpierw dodaj procedurę składowaną orders_set_auth_code do bazy danych:

```
-- Create orders_set_auth_code stored procedure
CREATE PROCEDURE orders_set_auth_code(IN inOrderId INT,
IN inAuthCode VARCHAR(50), IN inReference VARCHAR(50))
BEGIN
UPDATE orders
SET auth_code = inAuthCode, reference = inReference
WHERE order_id = inOrderId;
END$$
```

2. Dodaj metodę SetOrderAuthCodeAndReference() do klasy Orders w business/orders.php:

```
// Sets order's authorization code
public static function SetOrderAuthCodeAndReference ($orderId, $authCode,
$reference)
{
// Build the SQL query
$sql = 'CALL orders_set_auth_code(:order_id, :auth_code, :reference)';
// Build the parameters array
$params = array (':order_id' => $orderId,
':auth_code' => $authCode,
':reference' => $reference);
// Execute the query
DatabaseHandler::Execute($sql, $params);
}
```

3. Kodem do ustawienia tych wartości w bazie jest metoda SetOrderAuthCodeAndReference(), którą musimy dodać do klasy OrderProcessor w business/order_processor.php:

```
// Set order's authorization code and reference code
public function SetAuthCodeAndReference($authCode, $reference)
{
Orders::SetOrderAuthCodeAndReference($this->mOrderInfo['order_id'],
$authCode, $reference);
$this->mOrderInfo['auth_code'] = $authCode;
$this->mOrderInfo['reference'] = $reference;
}
```

Ten kod ustawia również odpowiednie elementy z tablicy $mOrderInfo, na wypadek gdyby były wymagane przed zakończeniem procesu OrderProcessor. W takiej sytuacji pobieranie tych wartości z bazy danych nie miałoby większego sensu, skoro wiemy już, jaki będzie wynik. W następnym rozdziale, kiedy będziemy zajmować się obsługą kart kredytowych, będziemy musieli ustawić dane w polach auth_code i reference w tabeli zamówień.

Ćwiczenie: Ustalanie daty wysyłki zamówienia

1. Po wysłaniu zamówienia należy zaktualizować datę wysyłki w bazie danych, która może być po prostu datą bieżącą. Dodaj procedurę składowaną orders_set_date_shipped do bazy danych tshirtshop:

```
-- Create orders_set_date_shipped stored procedure
```

```
CREATE PROCEDURE orders_set_date_shipped(IN inOrderId INT)

BEGIN

UPDATE orders SET shipped_on = NOW() WHERE order_id = inOrderId;

END$$
```

2. Dodaj nową metodę warstwy danych, SetDateShipped(), do klasy Orders w business/orders.php w następujący sposób:

```
// Set order's ship date

public static function SetDateShipped($orderId)

{

// Build the SQL query

$sql = 'CALL orders_set_date_shipped(:order_id)';

// Build the parameters array

$params = array (':order_id' => $orderId);

// Execute the query

DatabaseHandler::Execute($sql, $params);

}
```

3. Dodaj następującą metodę do klasy OrderProcessor w business/order_processor.php:

```
// Set order's ship date

public function SetDateShipped()

{

Orders::SetDateShipped($this->mOrderInfo['order_id']);

$this->mOrderInfo['shipped_on'] = date('Y-m-d');

}
```

Ćwiczenie: Wysyłanie e-maili do klientów i dostawców

1. Potrzebujemy dwóch metod obsługi wysyłania e-maili do klientów i dostawców. Dodaj metodę MailCustomer() do klasy OrderProcessor:

```
// Send e-mail to the customer

public function MailCustomer($subject, $body)

{

$to = $this->mCustomerInfo['email'];

$headers = 'From: ' . CUSTOMER_SERVICE_EMAIL . "\r\n";

$result = mail($to, $subject, $body, $headers);
```

```
if ($result === false)
{
throw new Exception ('Unable to send e-mail to customer.');
}
}
```

2. Dodaj metodę MailSupplier() do klasy OrderProcessor:

```
// Send e-mail to the supplier
public function MailSupplier($subject, $body)
{
$to = SUPPLIER_EMAIL;
$headers = 'From: ' . ORDER_PROCESSOR_EMAIL . "\r\n";
$result = mail($to, $subject, $body, $headers);
if ($result === false)
{
throw new Exception ('Unable to send e-mail to supplier.');
}
}
```

Ćwiczenie: Pobieranie szczegółów zamówienia i adresu klienta

1. Musimy pobrać ciąg reprezentujący zamówienie i adres klienta. Do tych zadań dodaj metodę GetCustomerAddressAsString() do klasy OrderProcessor, znajdującej się w business/ order_processor.php:

```
// Returns a string that contains the customer's address
public function GetCustomerAddressAsString()
{
$new_line = "\n";
$address_details = $this->mCustomerInfo['name'] . $new_line .
$this->mCustomerInfo['address_1'] . $new_line;
if (!empty ($this->mOrderInfo['address_2']))
$address_details .= $this->mCustomerInfo['address_2'] . $new_line;
$address_details .= $this->mCustomerInfo['city'] . $new_line .
$this->mCustomerInfo['region'] . $new_line .
$this->mCustomerInfo['postal_code'] . $new_line .
```

```
$this->mCustomerInfo['country'];
return $address_details;
}
```

2. Dodaj GetOrderAsString() do klasy OrderProcessor:

```
// Returns a string that contains the order details
public function GetOrderAsString($withCustomerDetails = true)
{
$total_cost = 0.00;
$order_details = '';
$new_line = "\n";
if ($withCustomerDetails)
{
$order_details = 'Customer address:' . $new_line .
$this->GetCustomerAddressAsString() .
$new_line . $new_line;
$order_details .= 'Customer credit card:' . $new_line .
$this->mCustomerInfo['credit_card']->CardNumberX .
$new_line . $new_line;
}
foreach ($this->mOrderDetailsInfo as $order_detail)
{
$order_details .= $order_detail['quantity'] . ' ' .
$order_detail['product_name'] . '(' .
$order_detail['attributes'] . ') $' .
$order_detail['unit_cost'] . ' each, total cost $' .
number_format($order_detail['subtotal'],
2, '.', '') . $new_line;
$total_cost += $order_detail['subtotal'];
}
// Add shipping cost
if ($this->mOrderInfo['shipping_id'] != -1)
```

```
{
$order_details .= 'Shipping: ' . $this->mOrderInfo['shipping_type'] .
$new_line;
$total_cost += $this->mOrderInfo['shipping_cost'];
}
// Add tax
if ($this->mOrderInfo['tax_id'] != -1 &&
$this->mOrderInfo['tax_percentage'] != 0.00)
{
$tax_amount = round((float)$total_cost *
(float)$this->mOrderInfo['tax_percentage'], 2)
/ 100.00;
$order_details .= 'Tax: ' . $this->mOrderInfo['tax_type'] . ', $' .
number_format($tax_amount, 2, '.', '') .
$new_line;
$total_cost += $tax_amount;
}
$order_details .= $new_line . 'Total order cost: $' .
number_format($total_cost, 2, '.', '');
return $order_details;
}
```

Jak to działa: modyfikacje procesora zamówień

Wprowadziłeś kilka zmian w klasach Orders i OrderProcessor, a także stworzyłeś sporo procedur przechowywanych w bazie danych. To cały kod infrastruktury, który obsługuje implementację potoku zamówień, co jest niezbędne w przypadku profesjonalnej witryny e-commerce.

Podsumowanie

Rozpoczęliśmy budowę szkieletu aplikacji i przygotowaliśmy go na lwią część funkcjonalności przetwarzania potoku zamówień, które zaimplementujemy w następnym rozdziale. W szczególności omówiliśmy następujące kwestie:

• Modyfikacje aplikacji TShirtShop, aby umożliwić nasze własne przetwarzanie potoku

• Podstawowe ramy dla naszego potoku zamówień

• Dodatki do bazy danych do kontroli danych i przechowywania dodatkowych wymaganych danych w tabeli zamówień