

# Wzorce projektowe w Java Card

także na tej platformie.

Tworzenie oprogramowania wymaga starannego projektu i dużej dokładności. Platforma Java Card do tej pory pozostawała swego rodzaju skansenem, gdzie szeroko akceptowany był kod, który w innych obszarach byłby absolutnie nieakceptowalny. Z pewnością warto więc poświęcić czas na zapoznanie się z dobrymi praktykami projektowania i programowania

## Dowiesz się:

- o dobrych praktykach programowania w Java Card;
- Jak dostosować istniejące wzorce projektowe do specyfiki platformy Java Card;
- Jak tworzyć kod Java Card, który można testować przy użyciu testów jednostkowych;

## Powinieneś wiedzieć:

- Znać podstawy programowania dla platformy Java Card;
- Posiadać wiedzę o projektowaniu oprogramowania;
- Znać język UML.

tów typu STK Handler w określonych momentach życia aplikacji STK,

- rozważne wykorzystanie globalnych i lokalnych zmiennych (tj. w szczególności: unikanie redundancji, minimalizacja liczby zmiennych lokalnych, minimalizacja liczby parametrów metod itd.),
- zakaz przechowywania referencji do tzw. *Entry Point Object*, jakim jest na przykład obiekt klasy APDU.

## Poziom trudności



pojawiają się czerwoną, pogrubioną czcionką już we wstępie do *Java Card™ & STK Applet Development Guidelines* – dokumentu dostarczanego przez jednego z producentów kart SIM.

## Wytyczne dla programistów Java Card

Najbardziej podstawowe zalecenia wymieniane we wspomnianych *Development Guidelines*, jakimi powinien kierować się programista Java Card, dotyczą zarówno kwestii ściśle powiązanych z samą technologią (modelem pamięci, programowaniem SIM Application Toolkit - STK), jak i ogólnie przyjętych zasad związanych z tworzeniem bezpiecznych aplikacji oraz czysto inżynierskimi praktykami tworzenia oprogramowania. Do rekomendacji wynikających wprost ze stosowania technologii Java Card można zaliczyć:

- stosowanie modyfikatora `static` dla zainicjalizowanych buforów danych (przechowujących na przykład teksty wyświetlane w komunikatach STK),
- obsługa współbieżności (tzw. *reentrance*) w STK, czyli zwrócenie uwagi, że sesja STK zainicjowana przez jedną aplikację STK może zostać przerwana na czas wykonywania sesji zainicjowanej przez inną aplikację,
- stosowanie się do zaleceń opisanych w dokumentach 3GPP 31.130 ((U)SIM API) i/lub TS 101476 (GSM API) dotyczących dostępności określonych obiektów

Z kolei, z punktu widzenia bezpieczeństwa tworzonego apletu, deweloperzy powinni stosować się do następujących zaleceń:

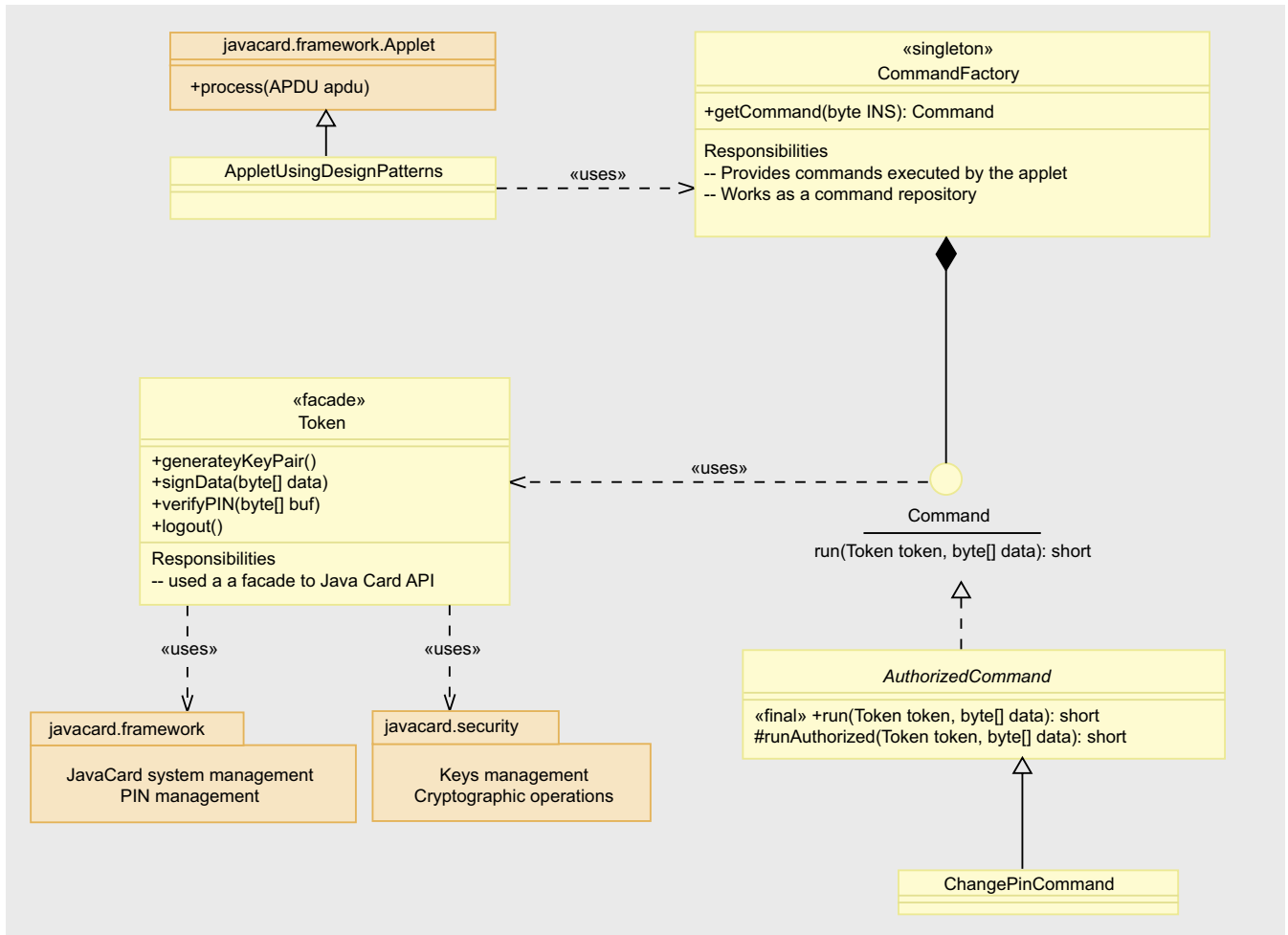
- unikanie przechowywania kluczy kryptograficznych oraz kodów PIN w tablicach prymitywnych typów, np. tablicy bajtów,
- ograniczenie czasu życia poufnych danych do czasu trwania sesji, w której są one wykorzystywane (np. poprawna obsługa tworzenia i kasowania kluczy sesyjnych w aplecie),
- przechowywanie poufnych danych w tablicach typu `transient`,
- ochrona danych przed atakiem typu `rollback`, związanym z występowaniem w API Java Card mechanizmu transakcji (po dokładny opis odsyłamy do *Java Card & STK Applet Development Guidelines*).

Pierwszym i najprostszym krokiem do zaspokojenia wyżej wymienionych wymagań jest z pewnością kierowanie się dobrymi praktykami programistycznymi. Zalecenia dla programistów Java Card są tutaj praktycznie identyczne jak dla inżynierów pracujących z wykorzystaniem innych technologii. Przede wszystkim podkreśla się konieczność:

- wykorzystywania metod API Java Card wszędzie, gdzie jest to możliwe, zamiast wprowadzania własnych implementacji,

Jak ważne jest właściwe prowadzenie projektów informatycznych ściśle po torach wyznaczanych przez metodologie tworzenia oprogramowania, wiadomo nie od dziś. Nawet dla najmniejszych komponentów, budujących większe systemy, niezbędne jest tworzenie modelu, który pozwala m.in.: na spojrzenie na kod z perspektywy wymagań systemu już od samego początku jego tworzenia, łatwiejszą weryfikację, testowanie oraz integrację poszczególnych części oprogramowania.

Podobnie jak istnienie projektu dla tworzonego oprogramowania, równie istotnym elementem procesu jego budowy jest sama jakość wytwarzanego kodu. Można przytoczyć szereg ogólnych rad, tzw. *best practices*, funkcjonujących w środowisku programistów, które pozwalają na osiągnięcie wysokiej jakości kodu. Stosowanie się do tych praktyk czy wytycznych przez deweloperów jest szczególnie ważne przy programowaniu w technologii Java Card. Możliwość aktualizacji apletu nagrałego na kartę i udostępnionego użytkownikowi są praktycznie żadne, a popelnianie błędów w implementacji może w najgorszym przypadku doprowadzić do nieodwracalnego uszkodzenia karty. Ostrzeżenia o konieczności dokładnego stosowania się do zaleceń, np. odnośnie użycia pamięci w Java Card,



Rysunek 1. Diagram klas apletu Java Card używającego wzorców projektowych

- tworzenia krótkich metod, co poprawia czytelność i pielęgnalność kodu,
- nie przekraczanie liczby 256 metod na klasę z uwagi na ograniczenia pamięci EEPROM oraz także ze względu na poprawność projektu apletu (należy mieć na względzie separację odpowiedzialności klas w aplecie).

rzyszywać, projektując i implementując oprogramowanie dla kart inteligentnych. W dalszej części artykułu przyjrzymy się, jakie wzorce można z powodzeniem stosować w apletach Java Card oraz jak wygląda ich implementacja,

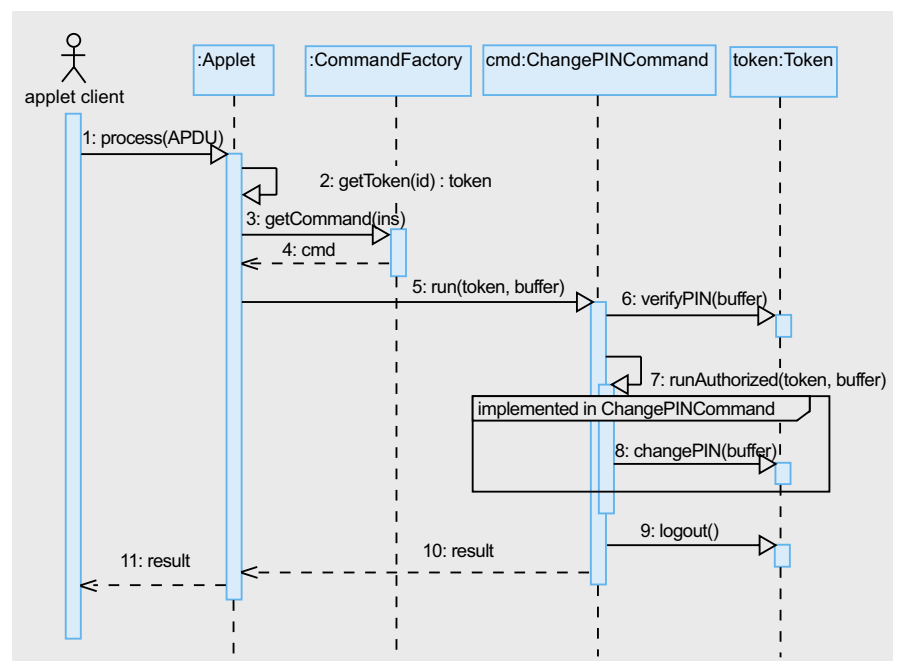
gdymy weźmiemy pod uwagę ograniczenia wynikające z zastosowania tej technologii.

Do przedstawienia kolejnych wzorców, jakie mogą być wykorzystane w apletach Java Card posłużymy, zaprezentowany na Rysunku 1, dia-

### Wzorce projektowe

Jednym z najistotniejszych elementów na drodze do osiągnięcia wysokiej jakości oprogramowania jest zastosowanie wzorców projektowych wszędzie tam, gdzie okazuje się, że określony problem można uogólnić do schematu zdefiniowanego przez określony wzorec. Pozwala to zarówno na utrzymanie prostego i zrozumiałego modelu systemu, jak i na wygodniejszą i prawdopodobnie lepszą implementację. Poza tym oszczędność czasu, jaką daje nam zastosowanie wzorców projektowych w porównaniu z *ponownym odkrywaniem koła*, ma istotny wpływ na koszty związane z całym projektem.

Czy wzorce projektowe można wykorzystać, programując w technologii Java Card? Pomimo sygnalizowanej już specyfiki tej technologii odpowiedź z pewnością jest twierdząca. Co więcej, należałoby powiedzieć, że wzorce projektowe nie tylko można, ale należy wyko-



Rysunek 2. Diagram sekwencji wywołania komendy zmiany kodu PIN

gram przygotowany w języku UML, ilustrujący niepełny projekt pewnego apletu. Niniejszy artykuł został podzielony według przyjętej ogólnie kategoryzacji, która wyróżnia wzorce *kreacyjne*, *strukturalne* oraz *behavioralne*. Wyróżnia się także kategorię wzorców wykorzystywanych w pro-

gramowaniu współbieżnym, jednak w technologii Java Card nie mają one zastosowania.

### Wzorce behawioralne

Jednym ze sposobów komunikacji z apilem Java Card jest komunikacja z wykorzystaniem

pakietów APDU (*Application Protocol Data Unit*). Zachowanie apletu jest wtedy determinowane poprzez wartość określonego bajtu (nazywanego w specyfikacji ISO7816-4 INS). Rozpowszechnioną praktyką (obecna niestety nawet w przykładach kodu udostępnianych przez producentów kart, takich jak Gemalto) jest użycie tego bajtu do budowy instrukcji switch w metodzie przetwarzającej apletu, która wywołuje dalej kod odpowiednich operacji. Dodatkowo, większość tych operacji przetwarza dane dostarczone w pakiecie APDU (w polu *Data*).

Tymczasem taka sytuacja doskonale nadaje się do wdrożenia *wzorca komendy*. Na podstawie przychodzącego do apletu APDU tworzony jest odpowiedni obiekt komendy (zgodnie z wartością bajtu INS), którego metoda *run* jest następnie wywoływana z odpowiednimi argumentami w celu dokonania przetwarzania. Diagram sekwencji takiego wywołania przedstawiony został na Rysunku 2.

Interfejs *Command* przedstawiony na diagramie klas z Rysunku 1 definiuje interfejs wszystkich komend, które ma do dyspozycji tworzony aplet. Rzeczywiste komendy implementują ten interfejs, definiując konkretne czynności, które są wykonywane w ramach danej komendy.

Wprawne oko zauważy jeszcze jeden wzorzec, który został wykorzystany przy budowie komend przykładowego apletu. Chodzi tutaj o wzorzec *template method*, który został wykorzystany do zapewnienia uwierzytelnienia dla metod dziedziczących po komendzie *AuthorizedCommand*. Komenda ta implementuje interfejs *Command* w ten sposób, że wywołuje najpierw metodę sprawdzania kodu PIN przekazanego w danych wejściowych, a następnie deleguje dalsze przetwarzanie do abstrakcyjnej metody *runAuthorized*, której implementacja jest dostarczana przez konkretne klasy dziedziczące po *AuthorizedCommand*. Dzięki temu, implementacja komend wymagających uwierzytelnienia jest dużo prostsza i nie wymaga powielania w każdej komendzie kodu sprawdzającego kod PIN. Kod źródłowy komend *AuthorizedCommand* oraz *ChangePinCommand* został zaprezentowany na Listingu 1 i 2. Warto zwrócić uwagę na elegancję oraz czytelność takiej implementacji, które uzyskane zostały dzięki zaproponowanemu podejściu.

### Wzorce kreacyjne

W poprzedniej sekcji omówiony został schemat, według którego można zastosować wzorzec komendy w aplicie Java Card. Jego naturalnym uzupełnieniem jest wykorzystanie *wzorca fabryki obiektów*, dzięki któremu możliwe jest oddzielenie logiki tworzenia komend od ich użycia.

Zgodnie ze specyfiką technologii Java Card, zastosowane fabryki muszą jednocześnie dzia-

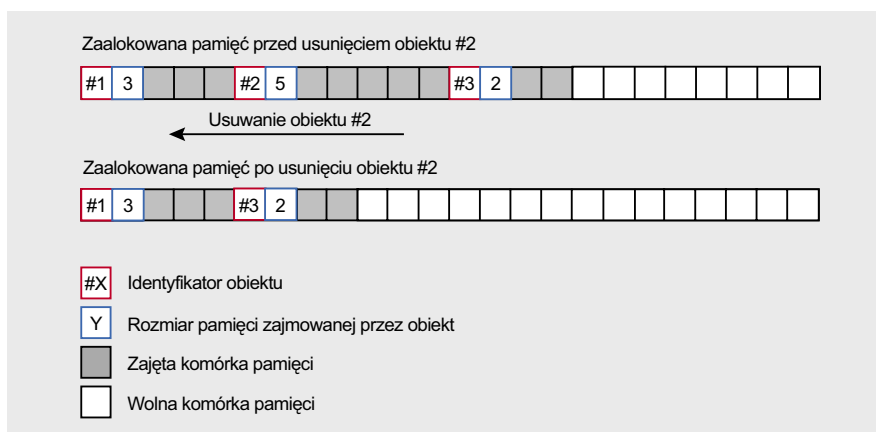
Listing 1. Klasa *AuthorizedCommand*

```
package pl.mobileexperts.sdj.samples;

/**
 * Root class for commands which need PIN authorization.
 */
public abstract class AuthorizedCommand implements Command {

    /**
     * Execute the command.
     *
     * @return number of bytes written to output buffer
     */
    public final short run(Token token, byte[] buffer, short length,
        short offset) {
        short size = 0;
        byte pinLength = buffer[(short) (offset + length - 1)];
        if (token.verifyPIN(buffer, (short) (offset + length - pinLength - 1),
            pinLength)) {
            try {
                size = runAuthorized(token, buffer,
                    (short) (length - pinLength - 1), offset);
            } finally {
                token.logout();
            }
        } else {
            ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
        }
        return size;
    }

    /**
     * Perform normal command functioning after PIN authorization.
     */
    protected abstract short runAuthorized(Token token, byte[] buffer,
        short length, short offset);
}
```



Rysunek 3. Organizacja i zarządzanie pamięcią typu TLV

łać jako *repozytoria obiektów* (tzw. *object pool*), gdyż z założenia, cała pamięć wymagana przez aplet powinna zostać zaalokowana podczas jego instalacji (we wspomnianych wyżej *Development Guidelines* znajduje się m.in. zalecenie: *ALL objects making up the application are created during installation*). Dlatego też fabryka komend (`CommandFactory`) budowanego apletu przechowuje referencje do wszystkich wykorzystywanych przez aplet komend i udostępnia mu je zgodnie ze wzorcem fabryki obiektów.

Wspomniane ograniczenia, wynikające z modelu zarządzania pamięcią w Java Card, wymuszają także, aby fabryka komend działała jako *singleton*. Szczególnie w przypadku, gdyby interfejs fabryki został rozszerzony o kolejne metody, bardzo wygodna jest możliwość pobrania referencji do fabryki z dowolnego miejsca kodu bez potrzeby odwoływania się do innych obiektów, które taką referencję mogłyby przechowywać.

## Wzorce strukturalne

Wzorec strukturalny, jaki zastosowany został w przykładowym aplecie, wynika niejako ze specyfiki technologii Java Card. Dostęp do operacji na kodach PIN, zarządzanie kluczami czy wykonywanie operacji kryptograficznych odbywa się przez niskopoziomowe API definiowane przez specyfikację Java Card (obiekty typu `OwnerPIN`, `Cipher` itp.) Aplikacje korzystające z apletu z reguły wykorzystują jedynie określoną część tych funkcjonalności. Dodatkowo, bezpośrednie wykorzystanie niskopoziomowego API zawsze zaciemnia kod i utrudnia jego analizę i późniejszą pielęgnację. Naturalnym rozwiązaniem dla tego typu sytuacji jest zastosowanie *wzorca fasady*, którego celem jest dostarczenie prostszego interfejsu dla bardziej skomplikowanych elementów czy bibliotek. W projekcie apletu założono, że obiekty klasy `Token` *przesłaniają* niskopoziomowe API Java Card służące m.in. do operacji przeprowadzania operacji kryptograficznych. Obiekty klasy `Token` mogą być utożsamiane z wirtualnymi i hermetycznymi miejscami na karcie inteligentnej, w których to przechowywany jest materiał kryptograficzny, i które pozwalają na wykonywanie określonych operacji (analogicznie do pojęcia tokenu zdefiniowanego w specyfikacji RSA PKCS#11 dla bibliotek kryptograficznych).

Komendy wywoływane przez aplet korzystają z fasady w postaci obiektów klasy `Token`, co pozwala na utrzymanie zwiezłości i przejrzystości ich kodu. Przykładowo, jak już pokazano wcześniej, zmiana kodu PIN to wywołanie odpowiedniej metody na obiekcie klasy `Token`, który to dopiero operuje już bezpośrednio na API Java Card. Takie podejście pozwala na dodatkową strukturalizację i hermetyzację poszczególnych elementów budujących aplet, co wpływa na poprawę jakości wytwarzanego kodu.

Innym przykładem zastosowania *wzorca fasady* w programowaniu Java Card jest wykorzystanie w kodzie apletu obiektu klasy dedykowanej do zarządzania pamięcią. Jak już zostało wspomniane wcześniej, programu-

jąc w Java Card spotykamy się z zaleceniem, aby cała pamięć potencjalnie wykorzystywana przez aplet była zaalokowana podczas jego instalacji. Należy również pamiętać o braku mechanizmu typu *garbage collector*. Zachodzi

### Listing 2. Klasa `ChangePINCommand`

```
package pl.mobileexperts.sdj.samples;
import javacard.framework.ISO7816;
import javacard.framework.ISOException;
import javacard.framework.Util;
/**
 * Command used to change PIN. Requires at least two bytes of data.
 * Last byte indicates new PIN length.
 */
public final class ChangePinCommand extends AuthorizedCommand {
    protected short runAuthorized(Token token, byte[] buffer, short length,
        short offset) {
        // at this moment current PIN was already authorized
        if (length < 2) {
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
        }
        // new PIN length
        byte pinLength = buffer[(short) (offset + length - 1)];
        if (pinLength < TokenConstants.MIN_PIN_SIZE ||
            pinLength > TokenConstants.MAX_PIN_SIZE) {
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
        }
        // check if received data has correct length
        if ((short) (length - 1) != Util.makeShort((byte) 0, pinLength)) {
            ISOException.throwIt(ISO7816.SW_WRONG_DATA);
        }
        // change PIN
        token.changePin(buffer, offset, pinLength);
        return 0;
    }
}
```

### Listing 3. Klasa `Token`

```
package pl.mobileexperts.sdj.samples;
import javacard.framework.ISO7816;
import javacard.framework.ISOException;
import javacard.framework.OwnerPIN;
/**
 * An abstraction of a cryptographic token, storing keys and PINs.
 */
public class Token implements TokenConstants {
    private boolean pinInitialized;
    private OwnerPIN pin;
    public Token() {
        pin = new OwnerPIN(MIN_PIN_SIZE, MAX_PIN_SIZE);
        // ...
    }
    public boolean verifyPin(byte[] buffer, short offset, byte length) {
        return pinInitialized && (pin.isValidated() ||
            pin.check(buffer, offset, length));
    }
    public void logout() {
        pin.reset();
    }
    // ...
}
```

więc potrzeba elastycznego dostępu i zarządzania pamięcią tak, aby była ona wykorzystana w sposób optymalny. Najbardziej efektywnym sposobem rozwiązania tego problemu jest stworzenie obiektu zarządcy pamięcią, co może być widziane także jako zastosowanie wzorca *puli zasobów*.

Przykładem implementacji takiego zarządcy jest wykorzystanie mechanizmu list TLV (*tag - length - value*). W ten sposób łatwo utrzymać ciągłość pamięci i przydzielać ją w optymalny sposób. Obiekty, które korzystają z zasobów pamięci, posługują się jedynie identyfikatorem (tzw. *tagiem*) obiektów

przechowywanych przez zarządcę pamięci, poprzez który zapisane dane są udostępniane. Zastosowanie list TLV zostało pokazane na przykładzie usuwania obiektu z pamięci na Rysunku 3.

## Czy warto?

Czy warto stosować wzorce projektowe, programując w technologii Java Card? Z pewnością tak! Oprócz zysków z punktu widzenia procesu wytwarzania systemów informatycznych, przy takim podejściu programuje się po prostu wygodniej i przyjemniej. Kod jest czytelny i przejrzysty, a znalezienie miejsca implementacji określonej funkcjonalności nie wymaga uważnego wpatrywania się w ekran podczas przewijania setek linii jednej wielkiej instrukcji *switch...* Cały aplet wygląda zgrabnie i, co najważniejsze, działa!

Poza aspektami estetycznymi pojawia się jeszcze kilka dodatkowych zalet. Po pierwsze, tak napisany kod apletu łatwo poddaje się testowaniu z wykorzystaniem bibliotek typu JUnit. Konieczność dokładnego testowania kodu jest równie istotna, jak wszystkie inne czynności podczas całego procesu tworzenia oprogramowania, o których była mowa wcześniej. Test komendy stworzonej dla przykładowego apletu jest widoczny we fragmencie Listingu 4. Takie testy pozwalają na stosowanie sprawdzonego i uznanego podejścia test-driven development i pokrycie testami większości logiki tworzonego apletu.

Jest oczywiste, że stosowanie dobrych inżynierskich zwyczajów, do jakich należy między innymi stworzenie odpowiedniego modelu systemu, używanie wzorców projektowych czy testowanie oprogramowania, jest nieodzownym elementem tworzenia oprogramowania wysokiej jakości. Co więcej, programowanie dla tak specyficznej platformy jak Java Card nie może być wymówką do zaprzestania stosowania tych praktyk! Niestety, nawet przykłady kodu prezentowane deweloperom przez producentów kart, zawierają koszarne instrukcje *switch* i kod zamknięty w jednej klasie... Pozostaje mieć nadzieję, że ten artykuł choć trochę przekona czytelników, że można osiągnąć duży przyrost jakości naprawde niewielkim nakładem pracy.

**Listing 4.** Test jednostkowy klasy *ChangePINCommand*

```
package pl.mobileexperts.sdj.samples;
import static org.easymock.EasyMock.*;
import static org.junit.Assert.*;
import javacard.framework.*;
import javacard.framework.*;
import org.junit.*;

public class ChangePinCommandTest {
    private Command command;

    @Before
    public void setUp() throws Exception {
        command = new ChangePinCommand();
        assertNotNull(command);
    }

    @Test
    public void testRun() {
        byte oldPinLength = (byte) 3;
        byte newPinLength = (byte) 4;
        byte[] buffer = new byte[] { (byte) 1, (byte) 1, (byte) 1, (byte) 1,
            newPinLength, (byte) 2, (byte) 2, (byte) 2, oldPinLength };
        // record expected behaviour
        Token token = createMock(Token.class);
        expect(
            token.verifyPin(buffer, (short) (newPinLength + 1),
                oldPinLength)).andReturn(true);
        token.changePin(buffer, (short) 0, newPinLength);
        expectLastCall();
        token.logout();
        expectLastCall();
        replay(token);
        // run and verify
        short result = command.run(token, buffer, (byte) buffer.length,
            (short) 0);
        assertEquals(result, (short) 0);
        verify(token);
    }
}
```

## W Sieci

- Strona domowa technologii Java Card: <http://java.sun.com/javacard/>;
- Bieżąca wersja specyfikacji Java Card 2.2.2: <http://java.sun.com/javacard/specs.html>;
- Dokumentacja on-line do Java Card 2.2.1: <http://www.cs.ru.nl/~woj/javacardapi221/>;
- Informacje dla deweloperów Java Card u producenta kart – Gemalto: <http://developer.gemalto.com/home/java-card.html>;
- Wskazówki dla deweloperów Java Card/STK: [http://developer.gemalto.com/fileadmin/contrib/downloads/pdf/Java\\_Card\\_STK\\_Applet\\_Development\\_Guidelines.pdf](http://developer.gemalto.com/fileadmin/contrib/downloads/pdf/Java_Card_STK_Applet_Development_Guidelines.pdf);
- (U)SIM Application Programming Interface (API); (U)SIM API for Java Card: <http://www.3gpp.org/ftp/Specs/html-info/31130.htm>
- USIM Application Toolkit (USAT): <http://www.3gpp.org/ftp/Specs/html-info/31111.htm>
- Dokumenty Stepping Stones organizacji SIMalliance i inne materiały dla deweloperów: <http://www.simalliance.org/>;
- Repozytorium wzorców projektowych dla języka Java: <http://www.javacamp.org/designPattern/>;
- Opis koncepcji list TLV: <http://en.wikipedia.org/wiki/Type-length-value>.

## O AUTORACH

Leszek Siwik, pracownik naukowy Katedry Informatyki Akademii Górniczo - Hutniczej, oraz Krzysztof Lewandowski i Adam Woś - architekci i zarządzający m.in. projektem mobilnego PKI realizowanym w Mobile Experts sp. z o.o. Wszyscy trzej specjalizują się w zagadnieniach szeroko pojętego oprogramowania mobilnego, a w szczególności tematyką wykorzystania urządzeń mobilnych w płatnościach, bankowości oraz do zapewnienia bezpieczeństwa informacji.