Doporučené postupy v programování

Návrh API

Proces návrhu · Obecné principy · Návrh tříd · Návrh metod

Lubomír Bulej

KDSS MFF UK

Poznámky:

Tyto slajdy čerpají primárně ze skvělé přednášku a slajdů o návrhu API a jeho důležitosti, kterou přednesl Joshua Bloch (dříve Sun, dnes Google) na JavaPolis 2005: How to Design a Good API & Why it Matters (slajdy).

V podstatě stejné poselství lze nalézt i v článku Michi Henninga v ACM Queue: API: Design Matters.

Zajímavá je také přednáška z konference JavaOne 2006 od Tima Boudreaua a Jaroslava Tulacha: How to write API that will stand the test of time. S tím souvisí pěkný tutorial o návrhu API na webu NetBeans: How to Design a (module) API a v neposlední řadě kniha Jaroslava Tulacha, z níž také čerpám – Practical API Design: Confessions of a Java Framework Architect, APress, 2008.

K čemu je API?

API = Application Programming Interface

Rozhraní existuje mezi alespoň dvěma subjekty

- projekty, subsystémy, třídy, ...
- tvůrci API a jeho uživatelé

Separace je hlavním důvodem pro vznik rozhraní

- oddělený překlad, oddělený vývoj, nezávislý vývoj ...
- oddělené úrovně abstrakce, oddělené povinnosti, ...
- "galvanické" oddělení zůčastněných stran

Rozhraní umožňuje odděleným částem komunikovat

- poskytuje stabilní kontrakt, který komunikaci umožňuje

Proč je API důležité?

API programů

- rozšiřitelnost, schopnost evoluce
- Firefox, Eclipse, ..., Emacs, Quake, ...

API platforem

- operační systémy
- jazykové runtimes (Java, .NET,...)
- jazyky samotné

API webových služeb

- Google Search, Google Maps
- ...

Poznámky:
K tématu rozšiřitelnosti má blízko (dlouhý a – v množství více než malém – filozofující) článek Steva Yeggea: The Pinocchio Problem.

Co je součástí API?

Vše, na co se může druhá strana spolehnout

- signatury metod, atributy tříd
- soubory, jejich umístění a obsah
- proměnné prostředí
- protokoly
  - přenos dat, clipboard, drag & drop
- chování
  - pořadí pro volání, zamykání, multithreading...
- lokalizace

Proč je návrh API důležitý?

Dobře navržené API je přínosem

- může přitáhnout a udržet uživatele
- vytváří závislost – intelektuální investice

Špatné navržené API je přítěží

- může uživatele odradit
- vyžaduje neustálou podporu
- kazí pověst společnosti

Je potřeba ho trefit na poprvé

- špatné věci nelze vzít zpět
- platí hlavně pro veřejná API

Příklad: rozhraní `Select()` v C#

.NET funkce pro čekání na socket

```
public static void Select (
  IList checkRead, IList checkWrite, IList checkError,
  int microseconds
);
```

C funkce pro čekání na socket

```
int select (
  int nfds,
  fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
  struct timeval *timeout
);
```

Poznámky:

Příklad převzat z článku Michi Henninga API: Design Matters.

Příklad: rozhraní `Select()` v C#

Představa použití C# rozhraní v kódu serveru

```
int timeout = ...;
ArrayList readList = ...; // sockets to monitor for reading
ArrayList writeList = ...; // sockets to monitor for writing
ArrayList errorList = ...; // sockets to monitor for errors

while (!done) {
  ArrayList checkRead = readList.Clone ();
  ArrayList checkWrite = writeList.Clone ();
  ArrayList checkError = errorList.Clone ();
  Select (checkRead, checkWrite, checkError, timeout);

  foreach (Socket socket in checkRead) {
    // deal with each socket ready for reading
  }

  foreach (Socket socket in checkWrite) {
    // deal with each socket ready for writing
  }

  foreach (Socket socket in checkError) {
    // deal with each socket that encountered an error
  }

  if (checkRead.Count == 0 && checkWrite.Count == 0 && checkError.Count == 0) {
    // no sockets are ready -- timed out...
  }
)
```

Poznámky:

- socketů může být mnoho, ale množiny se celkem nemění
- chyba nás většinou zajímá u socketů pro čtení nebo zápis
  - typicky se oba seznamy slučují, tady je navíc nutné eliminovat duplicitní sockety
- funkce nemá návratovou hodnotu a ničí předané parametry

- timeout je v mikrosekundách, max. timeout je asi 35 minut
  - není jasné jak čekat nekonečně dlouho

Příklad: rozhraní `Select()` v C#

Pomocná funkce pro test seznamů

```
private static boolean hasActiveSocket (
  IList readList, IList writeList, IList errorList
) {
  bool readListEmpty  = (readList  == null || readList.Count  == 0);
  bool writeListEmpty = (writeList == null || writeList.Count == 0);
  bool errorListEmpty = (errorList == null || errorList.Count == 0);

  return !readListEmpty || !writeListEmpty || !errorListEmpty;
}
```

Pomocná funkce pro kopírování seznamů

- rozhraní `IList` ani top-level objekt nemá `Clone()`
- aby měl objekt `Clone()`, musí implementovat `ICloneable`

Pomocná funkce pro slučování seznamů

- sloučí dva seznamy a eliminuje duplicity
- použití pro množinu socketů testovaných na chybu

Příklad: rozhraní `Select()` v C#

Wrapper funkce `doSelect()`

```
public static void doSelect (
  IList checkRead, IList checkWrite, IList checkError, int milliseconds
) {
  ArrayList readCopy;  // copies of the three parameters
  ArrayList writeCopy; // because Select() clobbers them
  ArrayList errorCopy;

  if (milliseconds <= 0) {
    // simulate waiting forever
    do {
      ... // copy socket lists
      Select (readCopy, writeCopy, errorCopy, Int32.MaxValue);
    } while (!hasActiveSocket (readCopy, writeCopy, errorCopy));

  } else {
    // handle finite timeouts
    int maxMilliseconds = Int32.MaxValue / 1000;

    int remaining = milliseconds;
    while ((remaining > 0) && !hasActiveSocket (readCopy, writeCopy, errorCopy)) {
      int timeout = milliseconds > maxMilliseconds ? maxMilliseconds : milliseconds;
      ... // copy socket lists
      Select (readCopy, writeCopy, errorCopy, timeout * 1000);
      remaining -= timeout;
    }
    ... // copy the three lists back to original parameters
}
```

Příklad: rozhraní `Select()` v C#

50-100 řádků boilerplate kódu

- wrapper + pomocné funkce
- důsledek drobných nedostatků v rozhraní

Drobné nedostatky v rozhraní `Select()`

- funkce přepisuje argumenty
- neumožňuje rozlišit timeout od změny stavu socketu
- neumožňuje čekat déle než 35 minut
- používá seznamy místo množin socketů

Vylepšené rozhraní funkce

```
public static bool Select (
  ISet checkRead, ISet checkWrite, Timespan timeout,
  out ISet readable, out ISet writable, out ISet error
);
```

Poznámky:
Hlavním problémem je, že funkce `Select()` představuje pouze wrapper pro low-level funkci systému. Ale kdyby jen to – nejenže selhává v nápravě špatně navrženého API z dob minulých, navíc do všechno přidává ještě vlastní snůšku problémů.

Jak se vás týká návrh API?

Pokud programujete, navrhujete API

- hranice mezi moduly, abstrakcemi, ...
- užitečné moduly jsou používány opakovaně
- jakmile máte uživatele, nemůžete API jen tak měnit

Je dobré myslet v intencích API

- vede k modularizaci a lepší architektuře
- i když ne všechna API jsou nutně veřejná

Co charakterizuje dobré API?

Je snadné se ho naučit i používat

- i bez dokumentace – intuitivní, regulární

Je těžké ho používat nesprávně

- nenechá uživatele dělat špatné věci

Je snadné porozumět klientskému kódu

- snižuje složitost a zvyšuje udržovatelnost kódu

Co charakterizuje dobré API?

Je dostatečně mocné na uspokojení požadavků

- ne však všemocné

Je jednoduše rozšiřitelné

- i když nebude na poprvé špatně, nebude úplně dobře
- pokud bude úspěšné, bude potřeba ho rozšiřovat

Je vhodné pro zamýšlené obecenstvo

- nelze udělat jedno správné API pro všechny
- uživatelé z různých domén mluví různými jazyky
- API představuje malý jazyk – musí odpovídat doméně

Rozšiřitelnost API vs. SPI

API - kód poskytuje službu

- rozšíření API spočívá v přidávání nových metod
- zpětně kompatibilní s kódem klientů

SPI - kód vyžaduje službu

- Service Provider Interface
- pluginové rozhraní pro různé implementace
- nelze přidávat metody, maximálně ignorovat staré

Nemíchat API a SPI

- různé způsoby použití, různí uživatelé
- komplikuje další vývoj rozhraní

Poznámky:
How to Design a (module) API.

Životní cyklus API

API se vyvíjí...

- spontánně
  - někdo vyvine nějakou věc, ta se zalíbí jinému, začne ji používat, začne chtít jiné věci – časem se kontrakt stabilizuje a vznikne API
- při návrhu
  - existuje potřeba stabilního kontraktu mezi dvěma subsystémy, API vznikne v důsledku **procesu** návrhu, po čase se vyvine a stabilizuje

Různá stádia vývoje API

- private, friend
  - typicky počáteční stav spontánního vývoje
- under development, stable, official
  - začátek řízeného návrhu
- deprecated

Proces návrhu API

Proces návrhu API

1. Zjistěte požadavky na API

- skeptický přístup – uživatelé neví, co chtějí
- místo požadavků na vás mohou chrlit řešení
  - tedy **jak** místo **co**
- cílem je dobrat se ke **skutečným** požadavkům
- ze skutečných požadavků zjistíte, co má rozhraní dělat
  - use cases, množina úloh, které má rozhraní vykonávat
- někdy může být jednodušší poskytnout obecnější rozhraní
  - pozor na přílišné zobecňování, viz. doporučení později

Proces návrhu API

2. Napište krátkou specifikaci

- 1 stránka stačí – úplnost specifikace není podstatná
  - důležitá je schopnost rychle dělat potřebné změny
  - krátká specifikace se mění snadno
- nechte specifikace zhodnotit co nejvíce lidem
  - snažte se brát vážně co říkají – naslouchejte
- až získáte jistotu, můžete dotáhnout specifikaci k úplnosti
  - bez psaní kódu to nepůjde

Proces návrhu API

3. Pište testovací kód proti API

- co nejdříve, co nejčastěji
- před tím, než API implementujete
  - nebudete muset zahazovat kód
- před tím, než API přesně specifikujete
  - nebudete muset zahazovat specifikaci
- během finalizace
  - ušetříte si nepříjemná překvapení
  - kód vám zůstane – příklady a unit testy
- ještě důležitější v případě SPI

- alespoň 3 různé pluginy před zveřejněním SPI

Proces návrhu API

4. Buďte realisté a počítejte s iterací

- nemůžete vyhovět všem
  - můžete nevyhovět všem stejnou měrou
- počítejte s tím, že budete dělat chyby
  - pár let reálného provozu si s nimi poradí
  - za přepokladu, že jste připraveni API rozvíjet

Pozor na *design by comittee*

- určete jednoho člověka zodpovědného za výsledný návrh
  - vzhledem k množství vstupů je lépe schopen udržet konzistenci
  - málokdy je rozhraní tak velké, aby to jeden člověk nezvládl

Příklad: thread-local proměnné

Pomocná třída pro thread-local proměnné

```
public final class ThreadLocal {
  private ThreadLocal () { /* non-instantiable */ }

  // Sets current thread's value for named variable.
  public static void set (String key, Object value);

  // Returns current thread's value for named variable.
  public static <T> T get (String key, Class <T> type);
}
```

Co je na rozhraní špatného?

- klíče představují sdílený globální prosto jmen
- možnost kolizí nebo falšování klíčů

Poznámky:

Příklad je převzat z přednášky Joshuy Blocha.

Příklad: thread-local proměnné

Pomocná třída pro thread-local proměnné

```
public class ThreadLocal {
  private ThreadLocal () { /* non-instantiable */ }

  public static class Key { Key() { } };

  // Generates unique, unforgeable key
  public static Key getKey () { return new Key (); }

  public static void set (Key key, Object value);
  public static <T> T get (Key key, Class <T> type);
}
```

Funguje, ale vyžaduje boilerplate kód

```
static ThreadLocal.Key serialNumberKey = ThreadLocal.getKey ();
ThreadLocal.set (serialNumberKey, nextSerialNumber ());
System.out.println (ThreadLocal.get (serialNumberKey));
```

Příklad: thread-local proměnné

Třída reprezentující thread-local proměnnou

```
public final class ThreadLocal <T> {
  public ThreadLocal () { }

  public void set (T value);
  public T get ();
}
```

Thread-local proměnná je klíčem

```
static ThreadLocal <Long> serialNumber = new ThreadLocal <> ();
serialNumber.set (nextSerialNumber ());
System.out.println (serialNumber.get ());
```

Obecné principy

Obecné principy

Myslete na uživatele.

Kdo jsou mí uživatelé?

- Co budou chtít s API dělat?
- Bude se jim API snadno používat? I bez dokumentace?
- Bude kód uživatelů dobře čitelný a spravovatelný?
- Je API interní nebo externí? Je potřeba více různých API?
- Je potřeba SPI?

Pohled implementátora je druhořadý

- je lepší investovat čas do jednoduchosti použití než implementace
- analogie k psaní kódu primárně pro pohodlí čtenáře, až poté písaře

Poznámky:

V blogovacím systému může být vhodné udělat zvlášť API pro autory obsahu a zvlášť pro čtenáře. Tyto dvě skupiny mají totiž úplně odlišné požadavky: Autor chce obsah číst i vytvářet, a pravděpodobně ho zajímá jen jeho blog, zatímco čtenář může obsah jen číst, ale zase ho pravděpodobně bude zajímat obsah více blogů, bude požadovat agregaci, apod.

Viz také API Design: The Principle of Audience (Ben Pryor)

Obecné principy

Minimalizujte údiv uživatele.

Snažte se uživatele nepřekvapit

- očekávání se těžko odhadují
- očekávání různých uživatelů konfliktní

"A user interface is well-designed when the program behaves exactly how the user thought it would." – Joel Spolsky

Poznámky:

Citát je převzat z Joelovy série článků o návrhu uživatelského rozhraní: 1, 2, 3, 4, 5, 6, 7, 8, 9.

V případě API je to podobně. Uživatel je "happy", když má věci pod kontrolou a dělají, co od nich očekává. Autor API (stejně jako autor UI) musí odhadnout "model uživatele" a přizpůsobit mu model programu (API).

Obecné principy

Dělejte jen jednu věc a dělejte ji dobře.

Funkce API by měla jít snadno vysvětlit

- pokud to nejde pojmenovat, něco je špatně
- počítejte s nutností rozdělovat a spojovat moduly

Obecné principy

Usilujte o co nejjednodušší *možné* řešení.

"Keep It Simple, Stupid." – anonym

"When in doubt, leave it out." – Joshua Bloch

"Everything should be made as simple as possible, but no simpler." – Albert Einstein

API by mělo být co nejmenší, ale ne menší

- důležité je, aby API splňovalo požadavky

- do rozhraní se dá snadno přidávat, ale nikdy odebírat
- konceptuální náročnost – jak dlouho potrvá se API naučit
- hledejte dobrý poměr síla/náročnost

Poznámky:

"Konstrukční dokonalosti není dosaženo tehdy, když už není co přidat, ale tehdy, když už nemůžete nic odebrat." – Antoine de Saint-Exupéry

Obecné principy

Implementace by neměla ovlivňovat API

- implementační detaily jsou matoucí a omezují možnost změny
- ujasněte si, co jsou v daném kontextu implementační detaily
  - pozor na příliš detailní specifikaci metod
- nenechte implementační detaily prosakovat do API

Minimalizujte přístupnost všeho

- maximalizuje skrývání informací, rozvolňuje vazby
- usnadňuje pochopení, testování a ladění

Obecné principy

Snažte se o typovou a běhovou konzistenci

- vše co jde zapsat by mělo správně fungovat
  - ne vždy je možné toho dosáhnout
- omezuje možnost nesprávného použití API

Příklad: java.sql

```
public interface Connection {
  ...
  public Savepoint setSavepoint ();
  public void rollback (Savepoint sp);
  ...
}

public interface Savepoint {
  public String getSavepointId ();
  public String getSavepointName ();
}
```

```
public interface Connection {
  ...
  public Savepoint setSavepoint ();
  ...

  public interface Savepoint {
    public void rollback ();
    public String getSavepointId ();
    public String getSavepointName ();
  }
}
```

Obecné principy

Na názvech záleží – API je malý jazyk

- samovysvětlující názvy, bez kryptických zkratek
- konzistentní používání stejných slov pro stejné věci

    - v rámci API, v rámci platformy
- usilujte o regularitu a symetrii
    - pozor, ne vždy dává smysl
- usilujte o dobrou čitelnost klientského kódu

```
if (car.speed () > 2 * SPEED_LIMIT) {
  generateAlert ("Watch out for cops!");
}
```

- pozor na změnu sémantiky při evoluci

Buďte otevření možnostem refaktoringu

- rozdělování/slučování modulů

Obecné principy

Dokumentujte ve velkém

- tutoriál, FAQ, reference

Dokumentujte v malém

- úplně každý jednotlivý prvek rozhraní
- kontrakty, vedlejší efekty, vlastnictví

Na dokumentaci záleží – pokud chybí

- uživatelé budou hádat nebo koukat do zdrojového kódu
- omezujete prostor pro opětovné použití kódu

Poznámky:

Pokud nutíte uživatele hádat, je to špatně. Pokud je nutíte koukat se do zdrojového kódu, je to ještě horší, protože je tím porušeno zapouzdření. Uživatelé nebudou programovat proti *rozhraní*, ale proti jeho jedné *konkrétní implementaci*. V tu chvíli ztrácíte flexibilitu ji někdy v budoucnu změnit.

Z důvodu větší flexibility implementace se vyplatí v dokumentaci příliš nepopisovat vnitřnosti – úroveň detailu by měla být právě dostačující k tomu, aby uživatel mohl s rozhraním pracovat.

Obecné principy

Přizpůsobte API cílové platformě

- využívejte idiomů cílové platformy
    - názvové konvence, standardní knihovny
    - napodobujte kód standardních knihoven
    - vyhněte se obsolete/deprecated věcem

- využívejte syntaktické prvky jazyka
  - parametry – defaulty, variabilní počet, "keyword params"
  - šablony/generiky, výčtové typy, ...

Pozor na multiplatformní/portovaná API

- pokud není nutná striktní kompatibilita, použijte idiomy cílové platformy

Návrh tříd (pro API)

Doporučení pro návrh tříd

Obecné zásady – detaily později

- omezte mutability, dědičnost používejte jen když to dává smysl
- navrhujte a dokumentujte pro dědičnost nebo ji zakažte

Usilujte o flexibilitu při zachování jednoduchosti

- žádné veřejné atributy – gettery/settery pro přístup k atributům
  - pozdní inicializace, synchronizace, přesun metody do nadtřídy, ...
- zvažte použití factory metody vs. konstruktoru
  - konstruktor lze jen přetěžovat, factory metody lze vhodně pojmenovat
  - možnost vytvářet instance podtříd, cachování instancí, synchronizace
  - Pozor: statické factory metody omezují testovatelnost

Doporučení pro návrh tříd

Zvažte použití vhodného nositele rozhraní

- interface – Java, ...
  - vícenásobná dědičnost, oddělení rozhraní a implementace, oddělení konceptů
  - problém s rozšiřováním pokud rozhraní implementuje klient – SPI
  - nemá konstruktor ani factory, může ho implementovat kdokoliv
  - není možné vynucovat sémantiku (co kdyby `String` byl interface?)
- final třída
  - podporuje různé úrovně přístupu, může poskytnout statické metody
  - lepší předpoklady pro evoluci – možno přidávat metody
- abstraktní třída?
  - může mít statické metody (ale to už interface také)
  - nepodporuje vícenásobnou dědičnost (signatur)
  - omezení přístupových práv (včetně konstruktoru)
    - poskytuje kontrolu nad tím, kdo může implementovat abstraktní metody

Poznámky:
How to Design a (module) API, Java API Design Guidelines.

Doporučení pro návrh tříd

Vyhněte se dědičnosti v SPI

- třídy a abstraktní třídy bohaté na virtuální metody
- málo dokumentované sémantické závislosti – implementační detail
- problém odpadá při použití final tříd a interfaces

Nahraďte třídy s virtuálními metodami kompozicí

- pomocí kombinace final tříd a (Java) interfaces
- dejte jednotlivým metodám v API jasný účel
  - metoda je určena k volání klientem
  - metoda představuje slot pro implementaci
  - metoda je určena k volání odvozenou třídou
- dejte jednotlivým typům v API jasný účel

Poznámky:

Viz. J. Tulach: Practical API Design, kapitola 10.

Velkým problémem v návrhu API stále zůstává dědičnost – specificky bohaté třídy se spoustou virtuálních metod a jejich implementacemi. Metody se mohou navzájem volat a třídy mohou tyto metody libovolně předefinovat a vytvářet sémantické závislosti, které jsou většinou považovány za implementační detail. Bez těchto detailů však téměř není možné korektně napsat odvozenou třídu a je nutné studovat zdrojový kód. Taková je např. situace v případě třídy `javax.swing.JComponent`. Nutnost číst zdrojový kód pro správné použití API však indikuje problematický návrh API samotného. V podstatě se dá říct, že v případě tříd s velkým množstvím různě provázaných virtuálních metod jsou problémy garantovány. Proto bývá nejlepší takové třídy z API eliminovat, cehož je možné docílit kombinací final tříd a interfaces.

Doporučení pro návrh tříd

Význam modifikátorů u metod v API

Modifikátory	Primární význam	Vedlejší významy
`public`	Metoda určena k volání externími klienty API.	Může být předefinována v odvozených třídách. Může bát volána z odvozených tříd.
`public abstract`	Metoda musí být implementována v odvozených třídách.	Může být volána externími klienty.

`public final`	Metoda určená pouze k volání.	Žádné.
`protected`	Metoda může být volána z odvozených tříd.	Může být předefinována v odvozených třídách.
`protected abstract`	Metoda musí být implementována v odvozených třídách.	Žádné.
`protected final`	Metoda může být volána z odvozených tříd.	Žádné.

Doporučení pro návrh tříd

Transformace metod s vedlejšími významy

Původní kód	Transformace

`public abstract void method ();`	`public final void method () {`
	`  methodImpl ();`
	`}`
	
	`protected abstract void methodImpl ();`

`public void method () {`
`  someCode ();`
`}`
	`public final void method () {`
	`  methodImpl ();`
	`}`
	
	`protected abstract void methodImpl ();`
	`protected final void someCode () {`
	`}`

`protected void method () {`
`  someCode ();`
`}`
	`protected abstract void method ();`
	`protected final void someCode () {`
	`}`

Příklad: kompozice jednoúčelových typů

Původní třída: API a SPI v jedné třídě

```
public abstract class MixedClass {
  private int counter;
  private int sum;

  protected MixedClass () {
    super ();
  }

  public final int apiForClients () {
    sum += toBeImplementedBySubclass ();
    return sum / counter;
  }

  protected abstract int toBeImplementedBySubclass ();

  protected final void toBeCalledBySubclass () {
```

```
      counter++;
    }
  }
```

Příklad: kompozice jednoúčelových typů

Nová třída: oddělené API a SPI

```
  public final class NonMixed {
    private int counter;
    private int sum;
    private final Provider provider;

    public interface Provider {
      public void initialize (Callback cb);
      public int toBeImplementedBySubclass ();
    }

    public static final class Callback {
      private NonMixed api;

      Callback (NonMixed api) {
        api = api;
      }

      public final void toBeCalledBySubclass () {
        api.counter++;
      }
    }
    ...
```

Příklad: kompozice jednoúčelových typů

Nová třída: oddělené API a SPI

```
    ...
    private NonMixed (Provider provider) {
      provider = provider;
    }

    public static NonMixed create (Provider provider) {
      NonMixed api = new NonMixed (provider);
      Callback callback = new Callback (api);
      provider.initialize (callback);
      return api;
    }

    public final int apiForClients () {
      sum += provider.toBeImplementedBySubclass ();
      return sum / counter;
    }
  }
```

Příklad: kompozice jednoúčelových typů

Použití nové třídy

```
  @Test public void useWithoutMixedMeanings () {
    class AddFiveMixedCounter implements NonMixed.Provider {
      private Callback callback;

      public int toBeImplementedBySubclass () {
        callback.toBeCalledBySubclass ();
        return 5;
      }

      public void initialize (Callback callback) {
        callback = callback;
      }
    }

    NonMixed add5 = NonMixed.create (new AddFiveMixedCounter ());
```

```
    assertEquals ("5/1 = 5", 5, add5.apiForClients ());
    assertEquals ("10/2 = 5", 5, add5.apiForClients ());
    assertEquals ("15/3 = 5", 5, add5.apiForClients ());
  }
```

Návrh metod (pro API)

Doporučení pro návrh metod

Obecné zásady – detaily později

- parametry a datové typy, počet a pořadí parametrů
- návratové hodnoty a výjimky
- přetěžujte opatrně – pozor na nejednoznačnosti
  - stejný počet parametrů, více parametrů stejného typu

Vynucujte dodržení kontraktu na rozhraní

- odolnost, robustnost, bezpečnost
  - uživatel nesmí uvést váš kód do nekonzistentního stavu
- více u defenzivního programování

Nenuťte uživatele dělat něco, co můžete sami

- snižuje potřebu psát boilerplate kód
  - otravné, cut & paste, náchylné k chybám

Příklad: nenuťte uživatele dělat zbytečnosti

Java: serializace XML dokumentu

```
import org.w3c.dom.*;
import java.io.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;

// DOM code to write an XML document to a specified output stream.
private static final
void writeDoc(Document doc, OutputStream out) throws IOException {
  try {
    Transformer t = TransformerFactory.newInstance()
      .newTransformer();
    t.setOutputProperty(
      OutputKeys.DOCTYPE_SYSTEM,
      doc.getDoctype().getSystemId()
    );
    t.transform(new DOMSource(doc), new StreamResult(out));
  } catch (TransformerException e) {
    throw new AssertionError(e); // Can't happen!
  }
}
```

Poznámky:

Příklad ukazuje na naprosto odbyté sbírání požadavků na API, protože serializace XML dokumentu je jedna z nejčastějších operací s XML/DOM API vůbec a v Javě je zcela zbytečně

komplikovaná. Vede to ke kopírování stále stejných kusů kódu po celém programu a tedy zanášení duplicit a chyb.

Příklad je převzat z přednášky Joshuy Blocha.

Doporučení pro návrh metod

Nechystejte žádná překvapení

- uživatele by nemělo překvapit chování metod
- stojí za větší úsilí při implementaci
- stojí za snížení výkonnosti

Příklad nesplněného očekávání v Javě

```
public class Thread implements Runnable {
  // Tests whether current thread has been interrupted.
  // Clears the interrupted status of current thread
  public static boolean interrupted ();
}
```

Poznámky:

K `Thread.interrupted()`: Tato metoda řekne, zda bylo vlákno přerušeno, ale zároveň příznak přerušení odnastaví. Další volání této metody budou tedy vždy vracet `false`. Přitom z názvu metody toto chování vůbec není zřejmé.

Viz také API Design: The Principle of Least Surprise.

Doporučení pro návrh metod

Selžete rychle

- o chybách dejte vědět co nejdříve po jejich vzniku
- pokud to jde tak při překladu
- za běhu pokud možno při prvním chybném volání

Příklad rychlého selhání

```
public class Properties extends Hashtable {
  public Object put (Object key, Object value);

  // Throws ClassCastException if this properties
  // contains any keys or values that are not Strings
  public void save (OutputStream out, String comments);
}
```