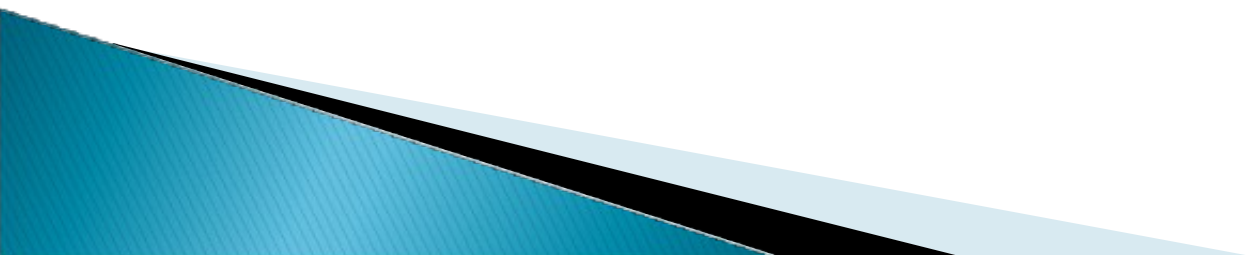


CGLib API

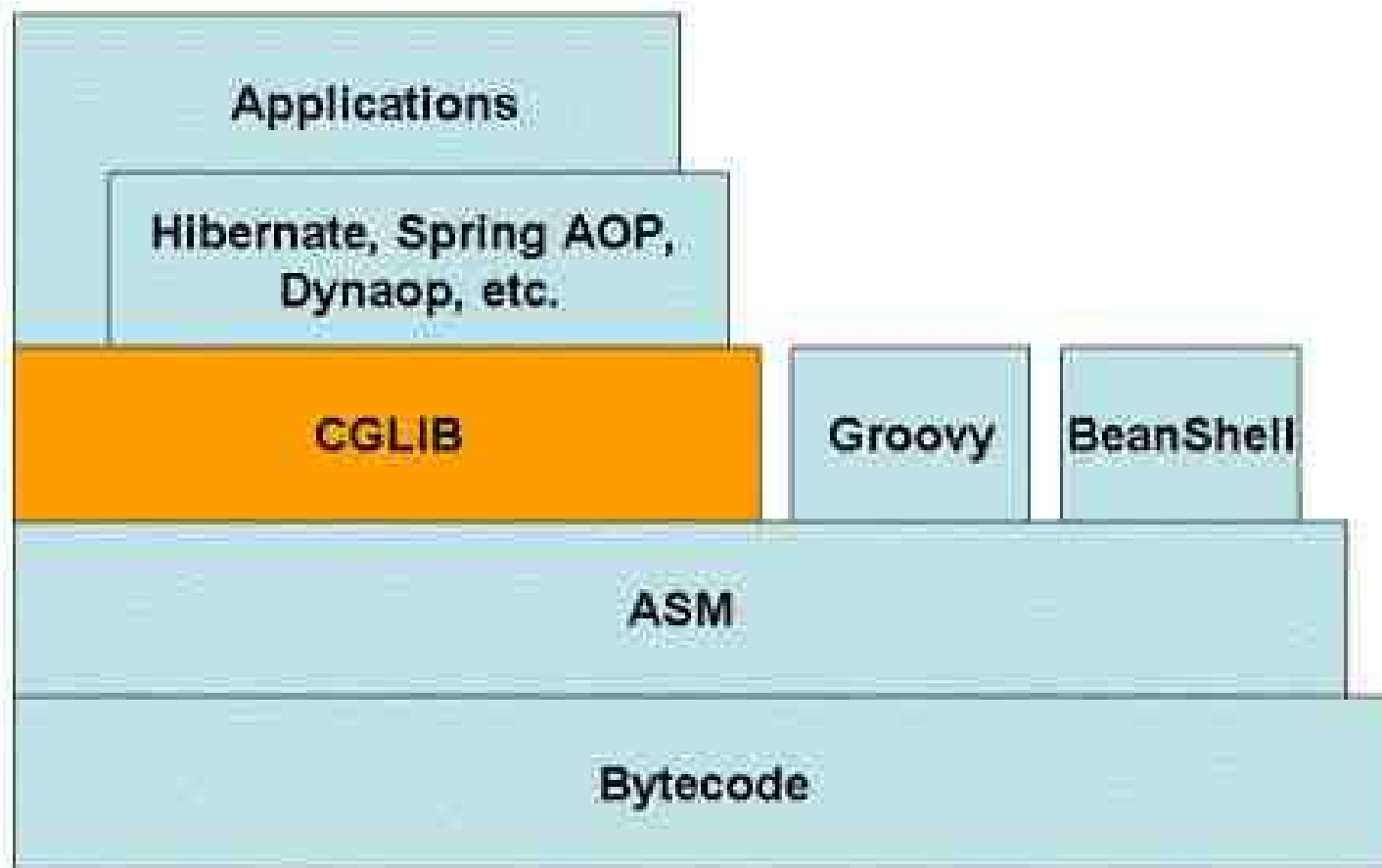
Maciej Jankowski



CGLib


- Biblioteka CGLib jest wysokopoziomową warstwą nad ASM.
- Używana do tworzenia proxy dla klas, które nie implementują interfejsów (ale nie tylko). Dzięki tej bibliotece, możemy dynamicznie tworzyć klasy, które nadpisują niefinalne metody.
- Biblioteka dostarcza również mechanizmy do modyfikacji bajecodu klasy na etapie ładowania przez classloader

Gdzie jest CGLib?




Source: <http://jnb.ociweb.com/jnb/jnbNov2005.html>

Najbardziej znane frameworki używające CGLib

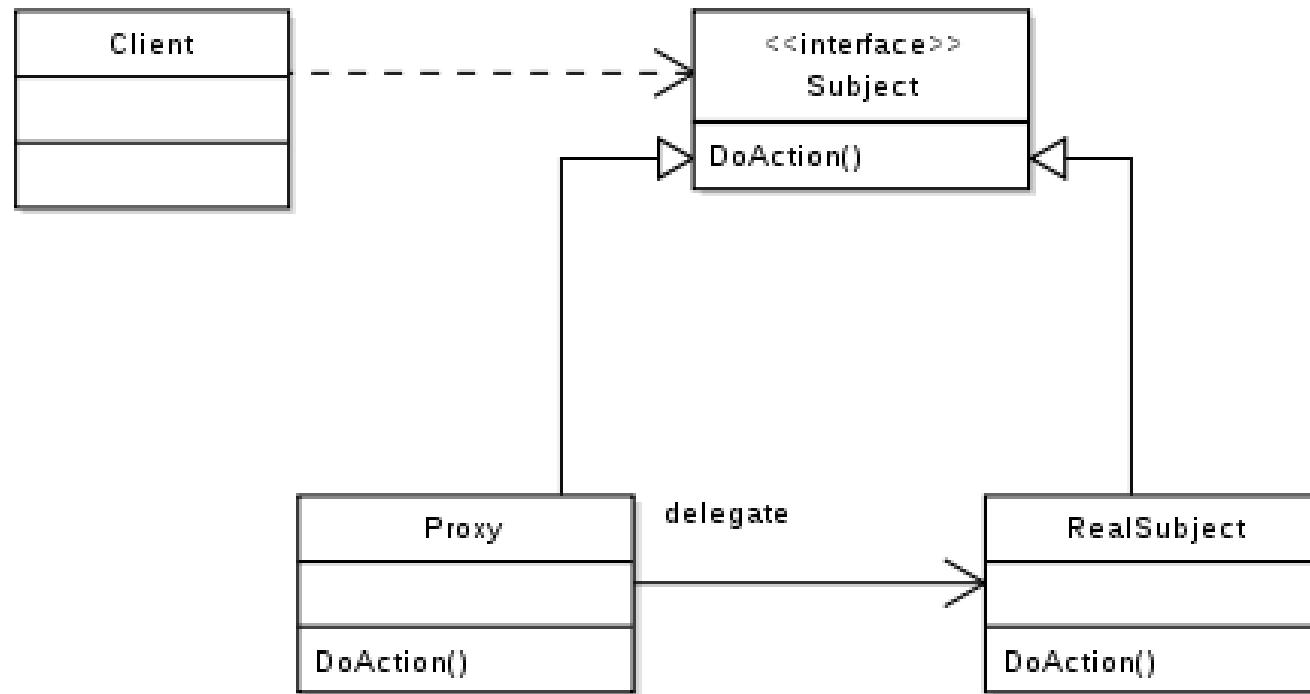
- Springframework (<http://www.springsource.org/>)
 - Hibernate (<http://www.hibernate.org/>) (do wersji 3.5.5)
 - EasyMock (<http://www.easymock.org/>)
- 

CGLib API

- `net.sf.cglib.proxy`
 - `net.sf.cglib.transform`
 - `net.sf.cglib.core`
 - `net.sf.cglib.reflect`
 - `net.sf.cglib.util`
 - `net.sf.cglib.beans`
- 

Część 1 - Proxy

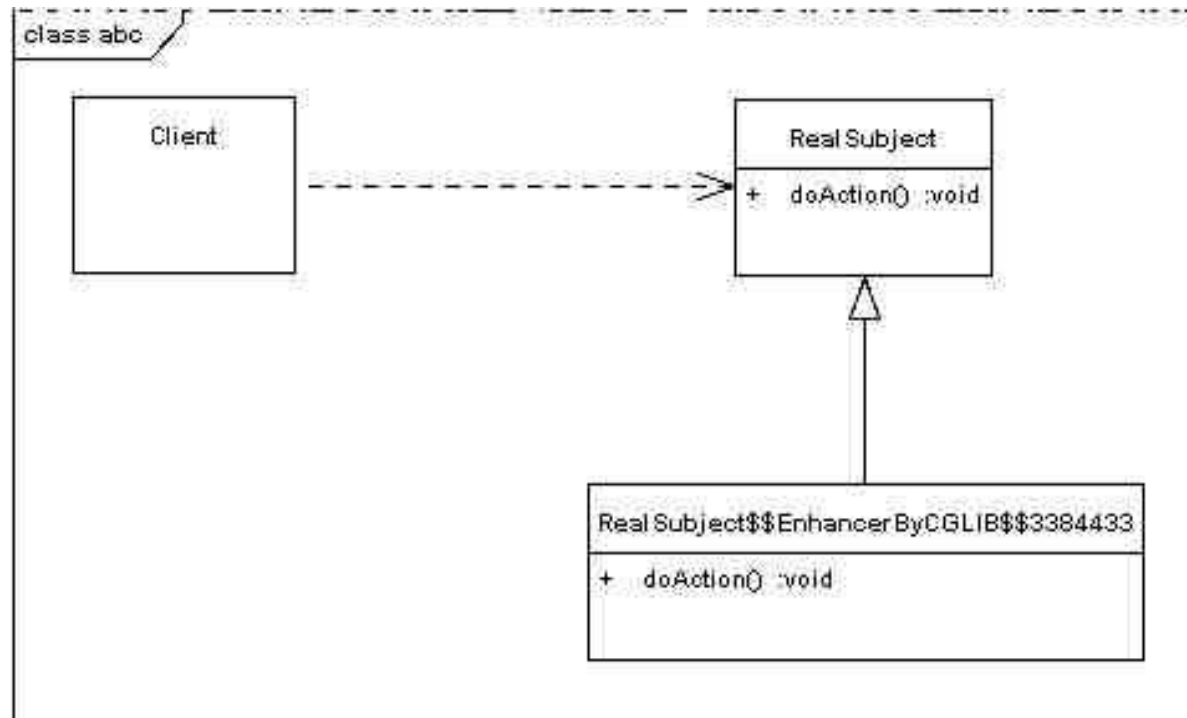
Proxy pattern



Source: http://en.wikipedia.org/wiki/Proxy_pattern

Jdk Proxy jest dokładną implementacją wzorca z powyższego diagramu. Widać od razu pewne ograniczenie - potrzebny jest interfejs.

Wsparcie dla tworzenia Proxy



- CGLib tworzy podklasę (lub implementuje interfejs). Za pomocą różnych mechanizmów dostarczonych przez bibliotekę, jesteśmy w stanie zdecydować jak wygląda implementacja metod w podklasie.

JDK Proxy vs CGLib Proxy

CGLib	Jdk Proxy
Nie potrzebuje interfejsu	Potrzebuje interfejs
Zależność od zewnętrznej biblioteki	Standardowa biblioteka javy
Stworzenie proxy polega na rozszerzeniu klasy dlatego nie możemy przechwycić wywołania metod finalnych.	Używamy delegacji, dlatego nie ma ograniczeń takich jak w CGLib

Podklasa

- Dodane zostały pola

Lnet/sf/cglib/proxy/MethodInterceptor; CGLIB\$CALLBACK_0

Ljava/lang/reflect/Method; CGLIB\$printHelloWjug\$0\$Method

Lnet/sf/cglib/proxy/MethodProxy; CGLIB\$printHelloWjug\$0\$Proxy


- Dodane zostały metody, które nadpisują metody bazowe

printHelloWjug()V


printHelloWorld()V

Implementacja metody **printHelloWjug()V** w podklasie

```
18 ifnull 37 (+19) (Callback is not set)
21 aload_0
22 getstatic #42 <pl/wjug/cglib/HelloClass$$EnhancerByCGLIB$
    $68a8d0de.CGLIB$printHelloWjug$0$Method>
25 getstatic #44 <pl/wjug/cglib/HelloClass$$EnhancerByCGLIB$
    $68a8d0de.CGLIB$emptyArgs>
28 getstatic #46 <pl/wjug/cglib/HelloClass$$EnhancerByCGLIB$
    $68a8d0de.CGLIB$printHelloWjug$0$Proxy>
31 invokeinterface #52 <net/sf/cglib/proxy/MethodInterceptor.intercept> count 5
36 return
37 aload_0
38 invokespecial #34 <pl/wjug/cglib/HelloClass.printHelloWjug>
41 return
```



CGLib Proxy API

- `net.sf.cglib.proxy.MethodInterceptor`
 - `net.sf.cglib.proxy.CallbackFilter`
 - `net.sf.cglib.proxy.FixedValue`
 - `net.sf.cglib.proxy.NoOp`
 - `net.sf.cglib.proxy.LazyLoader`
 - `net.sf.cglib.proxy.Dispatcher`
 - `net.sf.cglib.proxy.Mixin`
- 

CallbackFilter

- Pozwala zmapować metody podklasy wygenerowanej za pomocą CGLib na konkretne callbacki.
- Mapowanie zdefiniowane w CallbackFilter wpływa na wygenerowany bytecode i dlatego nie może zostać zmienione w czasie życia klasy.

Implementacja obiektu proxy

- Dodane zostały pola

`Lnet/sf/cglib/proxy/MethodInterceptor; CGLIB$CALLBACK_0`

`Lnet/sf/cglib/proxy/MethodInterceptor; CGLIB$CALLBACK_1`

- Implementacja metody `printHelloWjug()` w obiekcie proxy zawiera teraz odwołanie to pola `CGLIB$CALLBACK_0`, a implementacja metody `printHelloWorld()` odwołanie do pola `CGLIB$CALLBACK_1`

FixedValue

- Callback, który dla każdej metody zwraca taką samą wartość. Wartość ta jest definiowana w metodzie loadObject().
- Stosujemy, gdy chcemy mieć minimum narzutu czasowego.
- Wewnątrz callbacka nie mamy żadnej informacji o metodzie, która została wywołana, dlatego często łączymy FixedValue z CallbackFilter

CallbackFilterWithFixedValue 8.75 =====

MethodInterceptor 14.90 =====

NoOp

- Przekazuje wywołanie do obiektu stworzonego z nadklasy
- Szybsze niż wywołanie `proxy.invokeSuper()`, ponieważ decyzja nie jest podejmowana w czasie wykonania tylko w czasie kompilacji.
- Inaczej mówiąc nie używamy refleksji tylko zmieniamy bytecode
- Używamy razem z `CallbackFilter`

Performance I

- Nie ma dużej różnicy pomiędzy `InvocationHandler` a `MethodInterceptor`

`WithoutProxy` 55.5 ns

`JdkProxySum` 58.7 ns

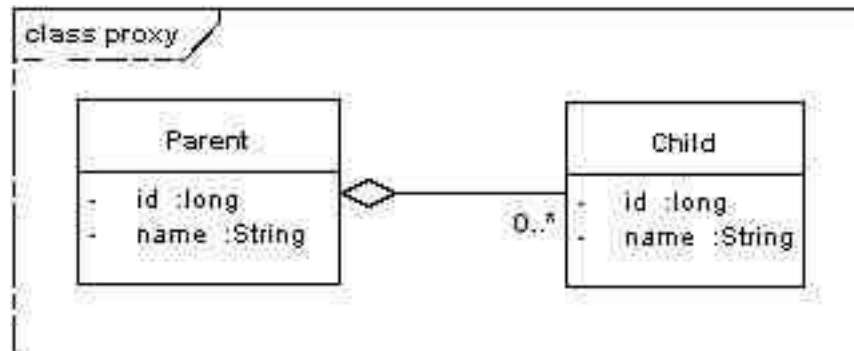
`CGLibProxySum` 57.2 ns

Performance II

- Załóżmy, że chcemy zmienić wynik jednej z metod klasy. Możemy to zrobić na dwa sposoby:
 - Użyć klasy MethodInterceptor (wolne)
 - Użyć kombinacji klas CallbackFilter, FixedValue, NoOp (szybkie)
- Testujemy następującą funkcjonalność:
 - Klasa bazowa zawiera trzy metody. Każda z nich zwraca string
 - Podklasa (proxy) zmienia wynik jednej z metod

MethodInterceptor 20.05 =====
NoOpCglib 3.69 =====

LazyLoader



- Sytuacja jak na powyższym diagramie
- W widoku chcemy pokazać tabelkę obiektów typu Parent
- Nie ma sensu z każdym obiektem typu Parent ładować listy obiektów children (np. z db przez sieć)
- Ładujemy listę obiektów children dopiero gdy jest ona potrzebna

LazyLoader

- Podpinamy callback LazyLoader do metody getChildren()
- Tworzymy podklasę klasy Parent przy użyciu CGLib
- Wywołania metod getId() oraz getName() zostają wywołane tak jak to zostało zdefiniowane w klasie Parent
- Gdy pierwszy raz użyjemy metody getChildren(), w metodzie loadObject() tworzony jest obiekt HeavyParent, który ładuje wszystkie obiekty Child będące w relacji z Parent
- Każde następne odwołanie do jakiejkolwiek metody parenta jest przekazane do obiektu stworzonego w metodzie loadObject()

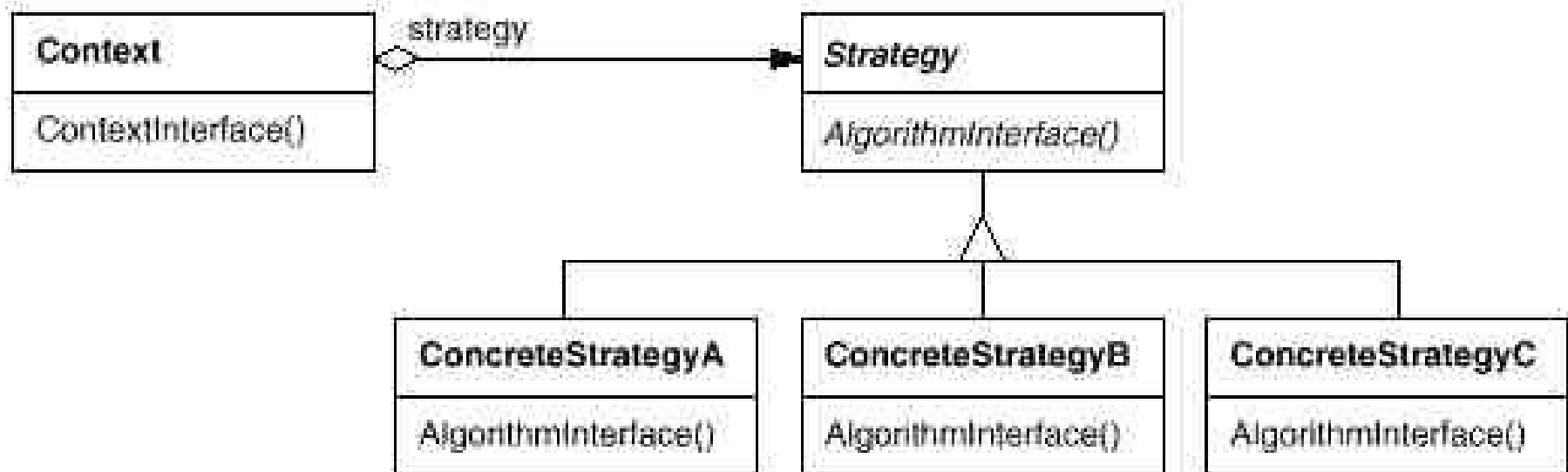
Dispatcher

- Pozwala przekierować wywołanie metody do innej instancji
- Ma taki sam interfejs jak LazyLoader. Różnica pomiędzy LazyLoader i Dispatcher jest taka, że w przypadku Dispatchera, loadObject() jest wykonywane przy każdym wywołaniu metody
- Możemy wykorzystać do trzymania referencji do sesji lub requestu

Mixin

- Pozwala połączyć wiele obiektów w jeden duży obiekt
- Przykład: Mając obiekt Car i obiekt Motorboat możemy stworzyć trzeci obiekt Amphibian, który potrafi pływać i jeździć

Strategia



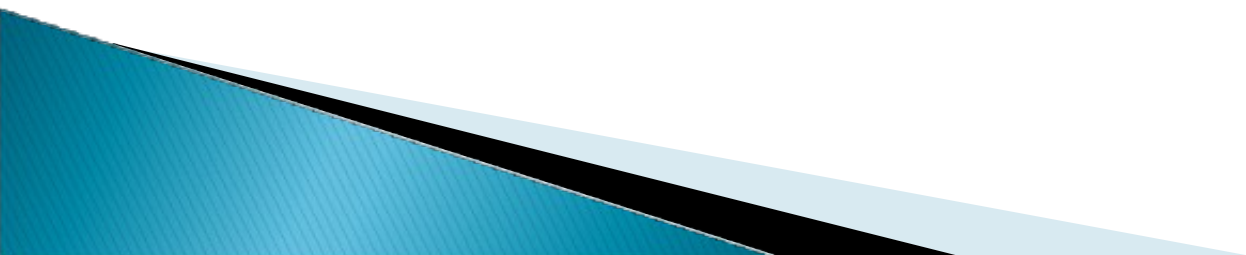
Strategia

- Definiując proxy za pomocą klasy Enhancer, możemy zażądać aby zostało ono zmodyfikowane za pomocą jednego z transformerów
- Rozszerzamy klasę DefaultGeneratorStrategy
- Istnieje gotowa strategia UndeclaredThrowableStrategy służąca do przechwycenia niezadeklarowanych wyjątków

UndeclaredThrowableStrategy

- Callback może rzucić sprawdzalny wyjątek, który nie był wymieniony w deklaracji metody klasy bazowej
- Klient stworzonego proxy nie spodziewa się tego wyjątku
- W tej sytuacji możemy użyć UndeclaredThrowableStrategy w celu przechwycenia tego rodzaju wyjątku i podmienienia go na inny wyjątek (na przykład niesprawdzalny)

Część 2 - Transformacje



ClassLoader I

Załadowanie klasy z dysku

```
java.io.InputStream is = //będziemy czytać plik .class z dysku  
ClassReader r = new ClassReader(is);  
ClassWriter w = new ClassWriter(0);  
r.accept(w, 0);  
byte[] b = w.toByteArray();  
return super.defineClass(name, b, 0, b.length, getDefaultDomain());
```

ClassLoader II

Modyfikacja klasy

```
ClassWriter w = new ClassWriter(0);
```

```
ClassVisitor v = new AddFieldAdapter(w, Opcodes.ACC_PUBLIC +  
Opcodes.ACC_FINAL + Opcodes.ACC_STATIC, "myAddedField",  
"Ljava/lang/String;");
```

```
v = new ChangeVersionAdapter(v);
```

```
r.accept(v, 0);
```

```
byte[] b = w.toByteArray();
```

```
return super.defineClass(name, b, 0, b.length, getDefaultDomain());
```

CGLib Transformations

- Transformacja jest wykonywana przez classloader o nazwie `TransformingClassLoader`
- Żeby zdecydować, czy dana klasa ma być zmodyfikowana dostarczamy klasę implementującą interfejs `ClassFilter`. Ten interfejs zawiera jedną metodę `accept(String className)`, która zwraca wartość typu `boolean`.
- Sama transformacja jest zdefiniowana w obiekcie implementującym podklasę klasy `ClassEmitterTransformer`. Ta implementacja nie musi być pojedynczym transformerem. Możemy użyć `ClassTransformerChain` aby zdefiniować ich dowolną ilość. Wykonają się w ustalonej przez nas kolejności.

CGLib Transformations API

- `net.sf.cglib.transform.impl.AddInitTransformer`
- `net.sf.cglib.transform.impl.AddStaticInitTransformer`
- `net.sf.cglib.transform.impl InterceptFieldTransformer`
- `net.sf.cglib.transform.impl.AddDelegateTransformer`
- `net.sf.cglib.transform.impl.AddPropertyTransformer`
- `net.sf.cglib.transform.ClassTransformerChain`

Podglądanie bytecodeu

- Jeżeli zdefiniujemy odpowiedni parametr do JVM, to CGLib będzie automatycznie zapisywał wygenerowane klasy we wskazanej lokalizacji
- (np. `-Dcglib.debugLocation=/tmp/cglib`)
- Jest to możliwe dzięki użyciu specjalnej implementacji ClassWriter-a o nazwie DebuggingClassWriter, Teraz wystarczy już tylko użyć jednego ze standardowych projektów do wyświetlania bytecodeu
- (np. <http://sourceforge.net/projects/jclasslib/>)
- Druga metoda polega na użyciu dekoratora dla interfejsu ClassVisitor o nazwie TraceClassVisitor

AddDelegateTransformer I

- AddDelegateTransformer daje nam możliwość oddelegowania wykonania części metod do innego obiektu.
- Przykład:

Mamy zdefiniowaną klasę User, która zawiera tylko pola. Chcemy dodać użytkownikowi dwie metody administracyjne: `changePassword(String)` i `shutdownSystem()`, które są zawarte w klasie AdminDelegate

CGLib pozwala tak zmodyfikować (etapie ładowania) klasę User, że zostanie ona wzbogacona o te dwie dodatkowe metody. Wywołanie każdej z nich na obiekcie klasy User spowoduje oddelegowanie wywołania do obiektu klasy AdminDelegate. Dodatkowo do klasy User musimy dodać interfejs zawierający odpowiednie deklaracje.

AddDelegateTransformer II

Jak została zmodyfikowana klasa User?

- Dodane zostało pole
`Lpl/wjug/cglib/transform/user/cglib/AdminDelegate;$CGLIB_DELEGATE`,
które zawiera referencję do instancji klasy AdminDelegate
- Dodane zostały dwie wcześniej wspomniane metody.

AddDelegateTransformer II

Poniżej znajduje się bytecode dla metody `changePassword(String)`

```
0 aload_0  
1 getfield #12 <pl/wjug/cglib/transform/user/cglib/User.$CGLIB_DELEGATE>  
4 aload_1  
5 invokevirtual #16 <pl/wjug/cglib/transform/user/cglib/AdminDelegate.changePassword>  
8 return
```

InterceptorFieldTransformer I

- Standardowo klasa User nie implementuje żadnego interfejsu, posiada tylko dwa pola i nie posiada żadnej metody. Jeżeli jednak na etapie ładowania pozwolimy na zmodyfikowanie klasy specjalnym transformerem o nazwie InterceptorFieldTransformer, to użytkownicy tej klasy będą mogli napisać następujący kod:

```
User u = new User();  
InterceptorFieldEnabled en = (InterceptorFieldEnabled) u;  
en.setInterceptorFieldCallback(new InterceptorFieldCallbackImpl("janek"));
```

- Inaczej mówiąc, będziemy w stanie przechwycić moment odwołania się do pola i wykonać dowolny fragment kodu

InterceptFieldTransformer II

Jak została zmodyfikowana klasa User?

- Dodane zostało pole

`Lnet/sf/cglib/transform/impl/InterceptFieldCallback$CGLIB_READ_WRITE_CALLBACK`
trzymające referencję do interceptora


- Dodane zostały (między innymi) dwie metody

`$cglib_read_name()Ljava/lang/String;`

`$cglib_write_name(Ljava/lang/String;)V`

Implementacja metody \$cglib_read_name()

```
0 aload_0
1 getfield #21 <pl/wjug/cglib/transform/user/cglib/User.name>
4 aload_0
5 invokeinterface #23
  <net/sf/cglib/transform/impl/InterceptFieldEnabled.getInterceptFieldCallback> count 1
10 ifnonnull 14 (+4)
13 areturn
14 astore_1
15 aload_0
16 invokeinterface #23
  <net/sf/cglib/transform/impl/InterceptFieldEnabled.getInterceptFieldCallback> count 1
21 aload_0
22 ldc #24 <name>
24 aload_1
25 invokeinterface #30 <net/sf/cglib/transform/impl/InterceptFieldCallback.readObject> count 4
30 checkcast #32 <java/lang/String>
33 areturn
```



InterceptorFieldTransformer III

- Za każdym razem gdy w kodzie odwołamy się do pola 'name', chcemy aby wywołana została metoda `$cglib_read_name()` lub `$cglib_write_name()`
- W jaki sposób zmusić obiekty aby zachowały się w ten sposób
- Okazuje się, że wszystkie klasy korzystające z klasy User są również modyfikowane przez transformer
- Inaczej mówiąc, jeżeli klasa odwołująca się do pola `User.name`, zostanie pominięta podczas transformacji, to żaden callback nie zostanie wywołany

InterceptorFieldTransformer IV

Implementacja metody TestUserClass.getUserName(User) pobierającej wartość pola User.name

- Przed modyfikacją

```
0 aload_1  
1 getfield #19 <pl/wjug/cglib/transform/user/cglib/User.name>  
4 areturn
```

- Po modyfikacji

```
0 aload_1  
1 invokevirtual #29 <pl/wjug/cglib/transform/user/cglib/User.$cglib_read_name>  
4 areturn
```

Inicjator statycznych pól danych

- CGLib pozwala dodać statyczny inicjator do klasy. Dzięki temu, za każdym razem gdy będziemy ładowali klasę wykona się zdefiniowany przez nas fragment kodu.
- StaticHook jest wywoływany w innym momencie niż standardowy blok static. W standardowym przypadku możemy nadpisać wartość pola, tutaj nie.

AddStaticInitTransformer I

Dodanie statycznego bloku kodu polega na wprowadzeniu dwóch modyfikacji do klasy User

1) Zmodyfikowana została metoda <cinit>

- Przed

```
0 bipush 30  
2 putstatic #12 <pl/wjug/cglib/transform/user/cglib/User.timeout>  
5 return
```

- Po

```
0 invokestatic #28 <pl/wjug/cglib/transform/user/cglib/User.CGLIB$STATICHOOK2>  
3 bipush 30  
5 putstatic #30 <pl/wjug/cglib/transform/user/cglib/User.timeout>  
8 return
```

AddStaticInitTransformer II

2) Dodana została statyczna metoda CGLIB\$STATICHOOK2()V

0 ldc #8 <pl.wjug.cglib.transform.user.cglib.User>

2 invokestatic #14 <java/lang/Class.forName>

5 invokestatic #20 <pl/wjug/cglib/transform/classloader/cglib/addinit/TestAddInit.init>

8 return



AddStaticInitTransformer III

- Zauważmy, że blok dodany za pomocą transformera powoduje inną modyfikację niż statyczny blok dodany za pomocą standardowych mechanizmów javy.
- Blok statyczny dodany transformerem jest wykonany przed przypisaniem wartości do pola
- Blok statyczny dodany za pomocą słowa kluczowego static jest wykonany po przypisaniu wartości do pola

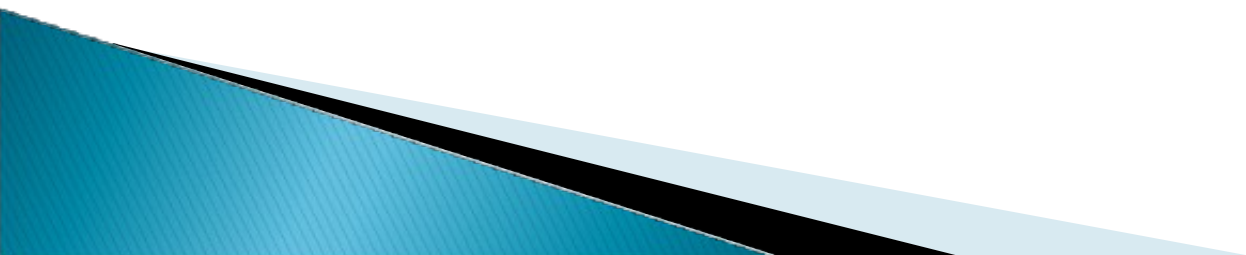
AddStaticInitTransformer IV

```
public String name = "adam";  
public static int timeout = 30;  
static{  
    timeout = 60;  
}
```

```
0 bipush 30  
2 putstatic #12 <pl/wjug/cglib/transform/user/cglib/User.timeout>  
5 bipush 60  
7 putstatic #12 <pl/wjug/cglib/transform/user/cglib/User.timeout>  
10 return
```

Łączenie transformerów

Podczas ładowania klasy możemy wykonać więcej niż jedną transformację, np. możemy dodać obiekt, do którego oddelegujemy wywołanie części metod oraz dodać statyczny inicjator



Alternatywne frameworki do generowania bytecodeu

- Javassist (www.javassist.org/) (high level)
- Jitescript (<https://github.com/qmx/jitescript>) (high level)
- ASM (<http://asm.ow2.org/>) (low level)
- BCEL (<http://commons.apache.org/bcel/>) (low level)
- Dexmaker (<http://code.google.com/p/dexmaker/>) (android)

Pytania