23 классических [шаблона проектирования](https://ru.wikipedia.org/wiki/%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F).

* [**Порождающие** шаблоны проектирования](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%80%D0%BE%D0%B6%D0%B4%D0%B0%D1%8E%D1%89%D0%B8%D0%B5_%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D1%8B_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)
  + [Abstract Factory — Абстрактная фабрика](https://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D0%B0%D1%8F_%D1%84%D0%B0%D0%B1%D1%80%D0%B8%D0%BA%D0%B0_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))
  + [Builder — Строитель](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D0%BE%D0%B8%D1%82%D0%B5%D0%BB%D1%8C_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))
  + [Factory Method — Фабричный метод](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B1%D1%80%D0%B8%D1%87%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))
  + [Prototype — Прототип](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%82%D0%BE%D1%82%D0%B8%D0%BF_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))
  + [Singleton — Одиночка](https://ru.wikipedia.org/wiki/%D0%9E%D0%B4%D0%B8%D0%BD%D0%BE%D1%87%D0%BA%D0%B0_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))
* [**Структурные** шаблоны проектирования](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%BD%D1%8B%D0%B5_%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D1%8B_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)
  + [Adapter — Адаптер](https://ru.wikipedia.org/wiki/%D0%90%D0%B4%D0%B0%D0%BF%D1%82%D0%B5%D1%80_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))
  + [Bridge — Мост](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D1%81%D1%82_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))
  + [Composite — Компоновщик](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%BE%D0%B2%D1%89%D0%B8%D0%BA_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))
  + [Decorator — Декоратор](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BA%D0%BE%D1%80%D0%B0%D1%82%D0%BE%D1%80_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))
  + [Facade — Фасад](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D1%81%D0%B0%D0%B4_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))
  + [Flyweight — Приспособленец](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D1%81%D0%BF%D0%BE%D1%81%D0%BE%D0%B1%D0%BB%D0%B5%D0%BD%D0%B5%D1%86_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))
  + [**Proxy** — Заместитель](https://ru.wikipedia.org/wiki/Proxy_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))
* [**Поведенческие** шаблоны проектирования](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B5_%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D1%8B_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)
  + [Chain of responsibility — Цепочка обязанностей](https://ru.wikipedia.org/wiki/%D0%A6%D0%B5%D0%BF%D0%BE%D1%87%D0%BA%D0%B0_%D0%BE%D0%B1%D1%8F%D0%B7%D0%B0%D0%BD%D0%BD%D0%BE%D1%81%D1%82%D0%B5%D0%B9)
  + [Command — Команда](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%B0_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))
  + [Interpreter — Интерпретатор](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BF%D1%80%D0%B5%D1%82%D0%B0%D1%82%D0%BE%D1%80_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))
  + [Iterator — Итератор](https://ru.wikipedia.org/wiki/%D0%98%D1%82%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))
  + [Mediator — Посредник](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%81%D1%80%D0%B5%D0%B4%D0%BD%D0%B8%D0%BA_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))
  + [Memento — Хранитель](https://ru.wikipedia.org/wiki/%D0%A5%D1%80%D0%B0%D0%BD%D0%B8%D1%82%D0%B5%D0%BB%D1%8C_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))
  + [Observer — Наблюдатель](https://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D0%B1%D0%BB%D1%8E%D0%B4%D0%B0%D1%82%D0%B5%D0%BB%D1%8C_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))
  + [State — Состояние](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%81%D1%82%D0%BE%D1%8F%D0%BD%D0%B8%D0%B5_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))
  + [Strategy — Стратегия](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D0%B0%D1%82%D0%B5%D0%B3%D0%B8%D1%8F_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))
  + [Template method — Шаблонный метод](https://ru.wikipedia.org/wiki/%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))
  + [Visitor — Посетитель](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%81%D0%B5%D1%82%D0%B8%D1%82%D0%B5%D0%BB%D1%8C_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))

**3. Intent of the Proxy design pattern**

С помощью паттерна проектирования «Заместитель» можно получить объект класса, обладающий функциональностью другого класса. Этот паттерн в GoF относится к числу структурных паттернов проектирования. Его описание в GoF гласит:  
«Обеспечивает заместитель для другого объекта с целью контроля доступа к нему.» Цель этого паттерна проектирования – предоставить класс, который бы обеспечивал окружающему миру доступ к функциональности другого класса.

**4.** **Proxy pattern describes how to solve such problems:**

Проблема:

* Необходимо контролировать доступ к объекту, не изменяя при этом поведение клиента.
* Необходимо иметь доступ к объекту так, чтобы не создавать реальные объекты непосредственно, а через другой объект, который может иметь дополнительную функциональность.

**5. Виды прокси**

1. **Remote proxy** – represent a remotely lactated object. To talk with remote objects, the client need to do additional work on communication over network. A proxy object does this communication on behalf of original object and client focuses on real talk to do.
2. **Virtual proxy** – delay the creation and initialization of expensive objects until needed, where the objects are created on demand. Hibernate created proxy entities are example of virtual proxies.
3. **Protection proxy** – help to implement security over original object. They may check for access rights before method invocations and allow or deny access based on the conclusion.
4. **Smart Proxy** – performs additional housekeeping work when an object is accessed by a client. An example can be to check if the real object is locked before it is accessed to ensure that no other object can change it.

**6-7. Структура (UML-диаграмма)**

Client

**Subject** – определяет интерфейс для реального объекта и его объекта прокси. Subject позволяет использовать прокси во всех местах, где используется реальный объект.

**RealSubject** – тот самый объект, создание которого занимает много времени, или находится на другой JVM. RealSubject реализует Subject и мы создаем его прокси, (реализация будет спратна за прокси).

**Proxy** – реализует Subject и содержит ссылку на RealSubject. Реализует дополнительную функциональность для контроля RealSubject объекта.

Вывод : Клиенты взаимодействуют с реальным объектом через его прокси посредством интерфейса.

**8-9. Proxy classes can implement many things (Real world example of proxy pattern)**

С применением прокси классов можно реализовать следующие вещи:

Без изменения кода, который находится в RealSubject. Следующий список лишь пример реалзиации, возможностей применения намного больше.

Логирование до и после работы метода,  
Допольнительная проверка аргументов  
Симулирование поведения оригинального класса  
Ленивая инициализация(виртуальный прокси)

Управление транзакциями   
Кеширование   
Обработка ошибок   
Мониторинг производительности

**Применимость**

**Ленивая инициализация (виртуальный прокси). Когда у вас есть тяжёлый объект, грузящий данные из файловой системы или базы данных.**

 Вместо того, чтобы грузить данные сразу после старта программы, можно сэкономить ресурсы и создать объект тогда, когда он действительно понадобится.

**Защита доступа (защищающий прокси). Когда в программе есть разные типы пользователей, и вам хочется защищать объект от неавторизованного доступа. Например, если ваши объекты — это важная часть операционной системы, а пользователи — сторонние программы (хорошие или вредоносные).**

 Прокси может проверять доступ при каждом вызове и передавать выполнение служебному объекту, если доступ разрешён.

**Локальный запуск сервиса (удалённый прокси). Когда настоящий сервисный объект находится на удалённом сервере.**

 В этом случае заместитель транслирует запросы клиента в вызовы по сети в протоколе, понятном удалённому сервису.

**Логирование запросов (логирующий прокси). Когда требуется хранить историю обращений к сервисному объекту.**

 Заместитель может сохранять историю обращения клиента к сервисному объекту.

**Кеширование объектов («умная» ссылка). Когда нужно кешировать результаты запросов клиентов и управлять их жизненным циклом.**

 Заместитель может подсчитывать количество ссылок на сервисный объект, которые были отданы клиенту и остаются активными. Когда все ссылки освобождаются, можно будет освободить и сам сервисный объект (например, закрыть подключение к базе данных).

Кроме того, Заместитель может отслеживать, не менял ли клиент сервисный объект. Это позволит использовать объекты повторно и здóрово экономить ресурсы, особенно если речь идёт о больших прожорливых сервисах.

In aspect oriented programming (AOP), an object created by the AOP framework in order to implement the aspect contracts (advise method executions and so on).

In hibernate, we write the code to fetch entities from the database. Hibernate returns an object which a proxy (by dynamically constructed by Hibernate by extending the domain class) to the underlying entity class. The client code is able to read the data whatever it needs to read with the proxy.

**10. Плюсы/Минусы**

**Плюсы:**

Позволяет контролировать сервисный объект незаметно для клиента.

**Минусы:**

* Усложняет код программы из-за введения дополнительных классов.
* Увеличивает время отклика от сервиса.

**11. Динамические прокси (Почему используют динамические прокси (SOLID принципы))**

In practical applications, the proxy class does not directly implement the functionality. Following the single responsibility principle, the proxy class does only proxying and the actual behavior modification is implemented in handlers. When the proxy object is invoked instead of the original object, the proxy decides if it has to invoke the original method or some handler. The handler may do its task and may also call the original method.

This is an advanced topic because it requires the use of the reflection class, or bytecode manipulation or compiling Java code generated dynamically. Or all of these. To have a new class not available as a bytecode yet during run-time will need the generation of the bytecode, and a class loader that loads the bytecode. To create the bytecode, you can use [cglib](https://github.com/cglib/cglib) or [bytebuddy](http://bytebuddy.net/) or the built-in Java compiler.

**12. Виды динамических прокси**

**Proxy JDK**

JDK Dynamic proxy can only proxy by interface (so your target class needs to implement an interface, which is then also implemented by the proxy class).

JDK proxies are implemented rather naively with only one interception dispatcher

Proxy.newProxyInstance, параметры вызова следующие:

* **ClassLoader**;
* **Массив типа** **Class**, должен принимать массив **интерфейсов**, которые реализует наш класс (**User**). **МЕТОДЫ ЭТИХ ИНТЕРФЕЙСОВ БУДУТ ПЕРЕХВАТЫВАТЬСЯ** (invocationHandler-ом).
* Экземпляр **InvocationHandler**, который будет **перехватывать** **методы** вызываемые для объекта **user** (*на самом деле, вызовы будут идти через вновь созданный****userProxy***).

На выходе получаем экземпляр некого класса(прокси), дающий следующую магическую функциональность :

* Исполняет все методы интерфейсов, переданных во 2-ом параметре на вход при вызове Proxy.newProxyInstance (в нашем примере это getName,setName,rename). В этом он похож на User;
* При вызове этих методов нашего экземпляра (*например userProxy.setName*) вызывается метод **INVOKE()** InvocationHandler-а. InvocationHandler уже дальше решает, как ему поступить —
  + вызвать соответствующий метод реального класса User
  + cделать что-то еще, в нашем случае

Здесь proxy — экземпляр прокси-класса, который может использоваться при обработке вызова того или иного метода. Второй параметр — method является экземпляром класса java.lang.reflect.Method. Значение данного параметра — один из методов, определенных в каком-либо из переданных при создании прокси-класса интерфейсов или их супер-интерфейсов. Третий параметр — массив значений аргументов метода.

То есть  в нашем примере **InvocationHandler** просто выводит имя вызываемого метода в консоль и вызывает его для сохраненного объекта. Таким образом, перед каждым исполнением методов user будет выводиться название вызванного метода.

[**InvocationHandler**](http://spec-zone.ru/RU/Java/Docs/7/api/java/lang/reflect/InvocationHandler.html) — интерфейс, реализованный обработчиком вызова экземпляра прокси. У каждого экземпляра прокси есть связанный обработчик вызова. Когда метод вызывается на экземпляр прокси, вызов метода кодируется и диспетчеризируется invoke метод его обработчика вызова.

Создается прокси-класс с помощью вызова метода Proxy.getProxyClass, который принимает класс-лоадер и массив интерфейсов (interfaces), а возвращает объект класса java.lang.Class, который загружен с помощью переданного класс-лоадера и реализует переданный массив интерфейсов.

Конструктор прокси-класса принимает один аргумент — реализацию интерфейса *InvocationHandler*. Соответственно, объект прокси-класса можно создать с помощью рефлексии, вызвав метод *newInstance* объекта класса *Class*. Однако, существует и другой способ — вызвать метод *Proxy.newProxyInstance*, который принимает на вход загрузчик классов, массив интерфейсов, которые будет реализовывать прокси-класс, и объект, реализующий *InvocationHandler*. Фактически, данный метод комбинирует получение прокси-класса с помощью *Proxy.getProxyClass* и создание экземпляра данного класса через рефлексию.

*Откуда же взялся этот «некий класс», экземпляр которого мы получили на выходе Proxy.newProxyInstance?*

Это динамически созданный класс, созданный ИЗ МАССИВА БАЙТ.  
**Цепочка вызовов :** Proxy.newProxyInstance -> Proxy.getProxyClass -> sun.misc.ProxyGenerator.generateProxyClass  
Этот последний метод возвращает массив байтов, который потом посредством ClassLoader.defineClass преобразуется в Class, и далее newInstance.  
В результате мы получаем программу которая генерирует сама себя.

**CGLIB**  
Классы в Java загружаются динамически во время выполнения. *Cglib*использует эту функцию языка Java, чтобы сделать возможным добавление новых классов в уже запущенную программу Java.

Byte code instrumentation allows to manipulate or to create classes after the compilation phase of a Java application. Since Java classes are linked dynamically at run time, it is possible to add new classes to an already running Java program. Hibernate uses cglib for example for its generation of dynamic proxies. Instead of returning the full object that you stored in a a database, Hibernate will return you an instrumented version of your stored class that lazily loads some values from the database only when they are requested. Spring used cglib for example when adding security constraints to your method calls.

У платформы java имеется две особенности. Для обеспечения кроссплатформенности программа сначала компилируется в промежуточный язык низкого уровня — байт-код. Вторая особенность загрузка исполняемых классов происходит с помощью расширяемых classloader. Это механизм обеспечивает большую гибкость и позволяет модифицировать исполняемый код при загрузке, создавать и подгружать новые классы во время выполнения программы.

*Hibernate*использует cglib для генерации динамических прокси. Например, он не вернет полный объект, хранящийся в базе данных, но вернет инструментальную версию хранимого класса, которая лениво загружает значения из базы данных по требованию.

Limitations of CGLIB proxies are:

Class for which a proxy is to be created must not be final

Code

An enhancer allows the creation of Java proxies for non-interface types.

The Enhancer dynamically creates a subclass of a given type but intercepts all method calls.

All generated proxied methods call this method instead of the original method. The original method may either be invoked by normal reflection using the Method object, or by using the MethodProxy (faster).

**Parameters:**

obj - "this", the enhanced object

method - intercepted Method

args - argument array; primitive types are wrapped

proxy - used to invoke super (non-intercepted method); may be called as many times as needed

CGLIB was designed and implemented more than ten years ago in AOP and ORM era.

In [2002](http://sourceforge.net/projects/cglib/) CGLIB defined a new standard to manipulate bytecode with ease. Many tools and methodology (CI, coverage, TDD, etc.) we have nowadays were not available or not mature at that time. CGLIB managed to be relevant for more than a decade ; that's a pretty decent achievement. It was fast and with an easy API to use than manipulating opcodes directly.

It defined new standard regarding code generation but nowadays it isn't anymore because environment and requirements have changed, so have the standards and goals.

CGLIB is quite fast, it is one of the main reason why it is still around

Note that mockito has replaced [CGLIB by Byte Buddy](https://github.com/mockito/mockito/pull/171) in version 2.1.0.

**Javaasist**

This library comes with a compiler that takes strings containing Java source code which are translated into Java byte code during the runtime of an application.

CGLIB (and javassist) can create a proxy by subclassing. In this scenario the proxy becomes a subclass of the target class. No need for interfaces.

* Javassist (Java programming assistant)
* Javassist provides two levels of API: source level and bytecode level.

Using the source-level API, programmers can edit a class file without knowledge of the specifications of the Java bytecode

Using the bytecode-level API allows the users to directly edit a class file as other editors.

* Javassist is recognized to be slower than Cglib

 ProxyFactory which is the equivalent of the CGLIB's Enhancer

Also Javassist is recognized to be slower than Cglib. This is mainly due to its approach of reading class files instead of reading loaded classes such as CGLIB does.

**Generating a Java Class**  
**Adding Fields to Existing Class Bytecode**  
**Adding Constructor to Class Bytecode**

## Byte Buddy

Byte Buddy is a rather new library but provides any functionality that CGLIB or Javassist provides and much more. Byte Buddy can be fully customised down to the byte code level and comes with an expressive domain specific language that allows for very readable code.

* It supports all JVM bytecode versions, including Java 8 semantic changes of some opcodes regarding default methods.
* Highly configurable
* Quite fast ([benchmark](http://zeroturnaround.com/rebellabs/testing-the-performance-of-4-java-runtime-code-generators-cglib-javassist-jdk-proxy-byte-buddy/) [code](https://github.com/raphw/byte-buddy/tree/master/byte-buddy-benchmark/src/main/java/net/bytebuddy/benchmark))
* Type safe fluent API
* Type safe callbacks
* Very well documented
* Lots of example
* Clean code, ~94% test coverage
* Android DEX support

The main downside perhaps, would the API is a bit verbose for a beginner but it is designed as an opt-in API shaped as a proxy generation DSL ; there's no magic or questionable defaults. When manipulating byte code it is probably the most safe and the most reasonable choice. Also with multiple examples and a big tutorial this is not a real issue.

In October 2015 this projects received the [Oracle Duke's choice award](https://www.oracle.com/corporate/pressrelease/dukes-award-102815.html). At this times it just reached the [1.0.0 milestone](https://github.com/raphw/byte-buddy/blob/master/release-notes.md), which is quite an achievement.

Note that mockito has replaced [CGLIB by Byte Buddy](https://github.com/mockito/mockito/pull/171) in version 2.1.0.

instruction is method which allows us to select any number of methods that we want to override.

instruction intercept determines the implementation that should override all methods of the given selection.

The InvocationHandlerAdapter allows to use existing InvocationHandlers from the for the [proxy classes](http://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Proxy.html) that ship with the Java Class Library.

getLoaded method returns an instance of a Java Class that represents the dynamic class which is now loaded

**16. Performance Comparison of JDK Proxy, cglib, Javassist and Byte Buddy**

The first benchmark in the above table measures a library's runtime for subclassing Object without implementing or overriding any methods. This gives us an impression of a library's general overhead in code generation. In this benchmark, Java proxies perform better than other libraries due to optimizations that are only possible when assuming to always extend an interface. Byte Buddy also checks classes for generic types and annotations what causes additional runtimes. This performance overhead is also visible in the other benchmarks for creating a class.

Benchmark (2a) shows the measured runtime for creating (and loading) a class that implements a single interface with 18 methods,

(2b) shows the execution time for the methods generated for this class.

Similarly, (3a) shows a benchmark for extending a class with the same 18 methods which are implemented.

Byte Buddy provides two benchmarks, due to a possible optimization that is possible for an interceptor that always executes the super method. Sacrificing some time during class creation, the execution time of a Byte Buddy-created classes typically reaches the baseline, meaning that the instrumentation creates no overhead at all. It should be noted that Byte Buddy outperforms any other code generation library also during class creation, if the meta data processing was disabled. As the runtime of code generation is however so minimal compared to a program's total runtime, such an opt-out is not available as it would gain very little performance to the sacrifice of complicating the library code.

You might have noticed that Byte Buddy is listed with two measurements where the second italic numbers is significantly larger. Both numbers represent different approaches of implementing a super method invocation.