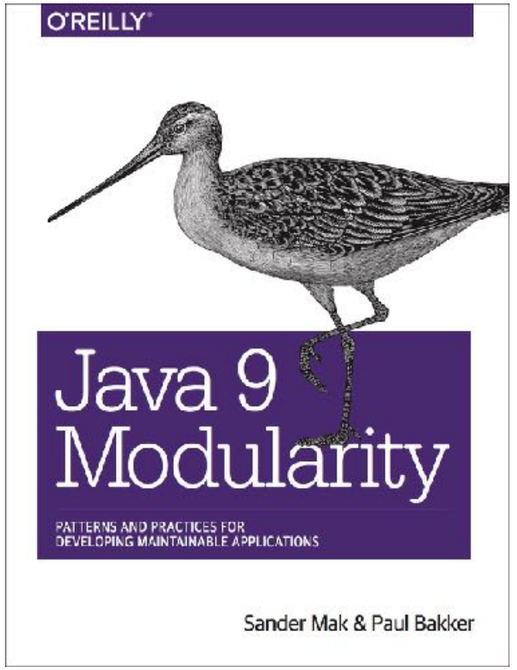
JPoint

luminis
Conversing worlds



Designing for Modularity With Java Modules

By Sander Mak

@Sander_Mak

Today's journey

Module primer

Services & DI

Modular design

Layers & loading

Designing for Modularity with Java 9

What if we ...

... forget about the classpath

... embrace modules

... want to create truly modular software?

Designing for Modularity with Java 9

What if we ...

... forget about the classpath

... embrace modules

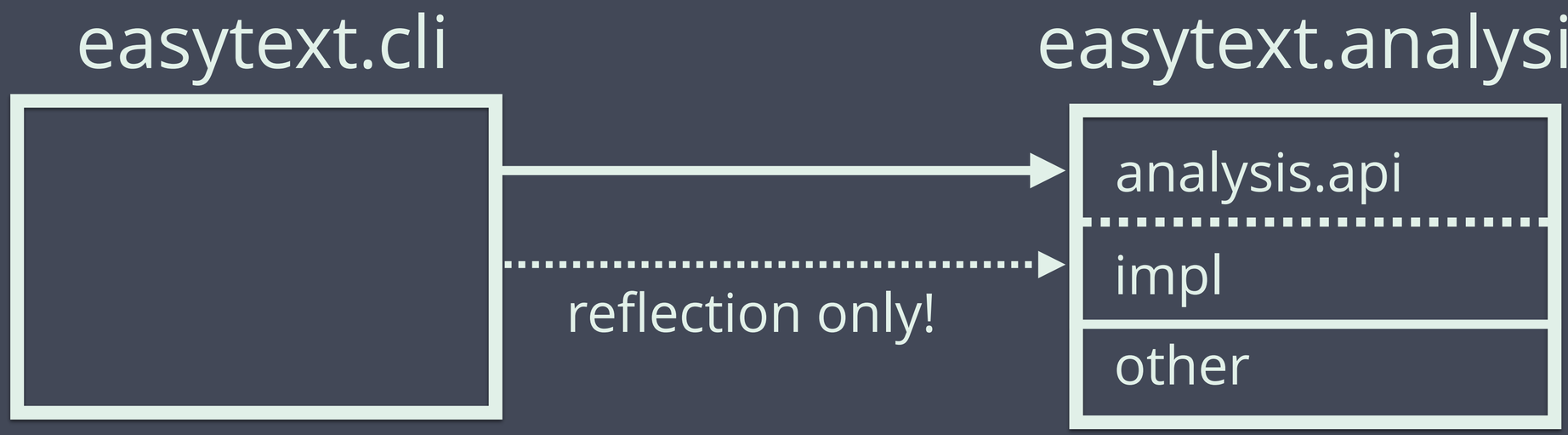
... want to create truly modular software?

Design **Constraints** **Patterns**

A Module Primer

```
module easytext.cli {
  requires easytext.analysis;
}
```

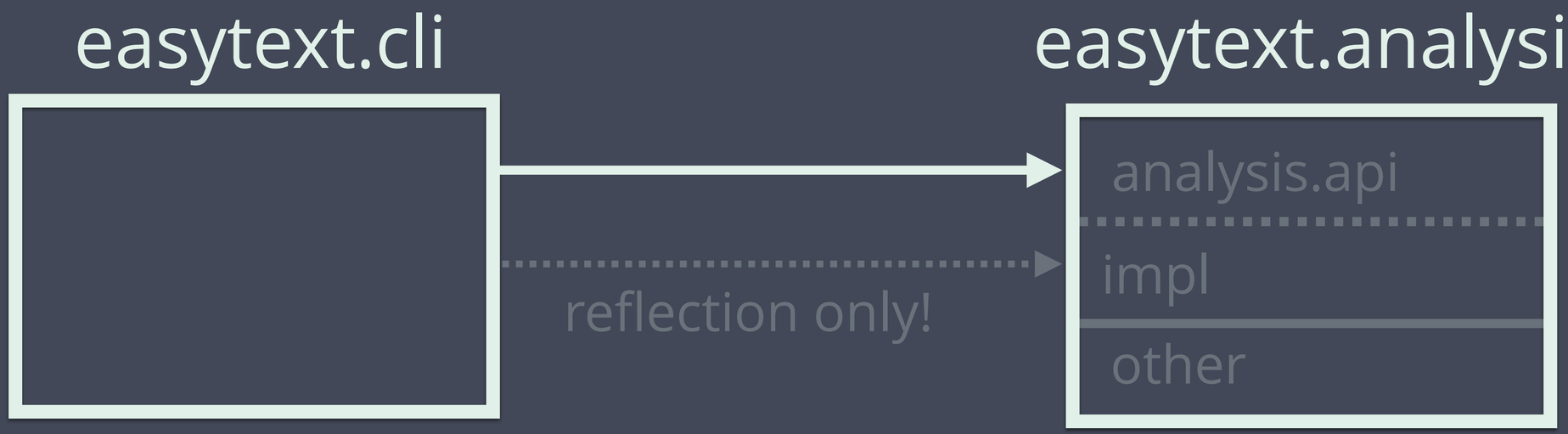
```
module easytext.analysis {
  exports analysis.api;
  opens impl;
}
```



A Module Primer

```
module easytext.cli {
  requires easytext.analysis;
}
```

Modules define dependencies explicitly

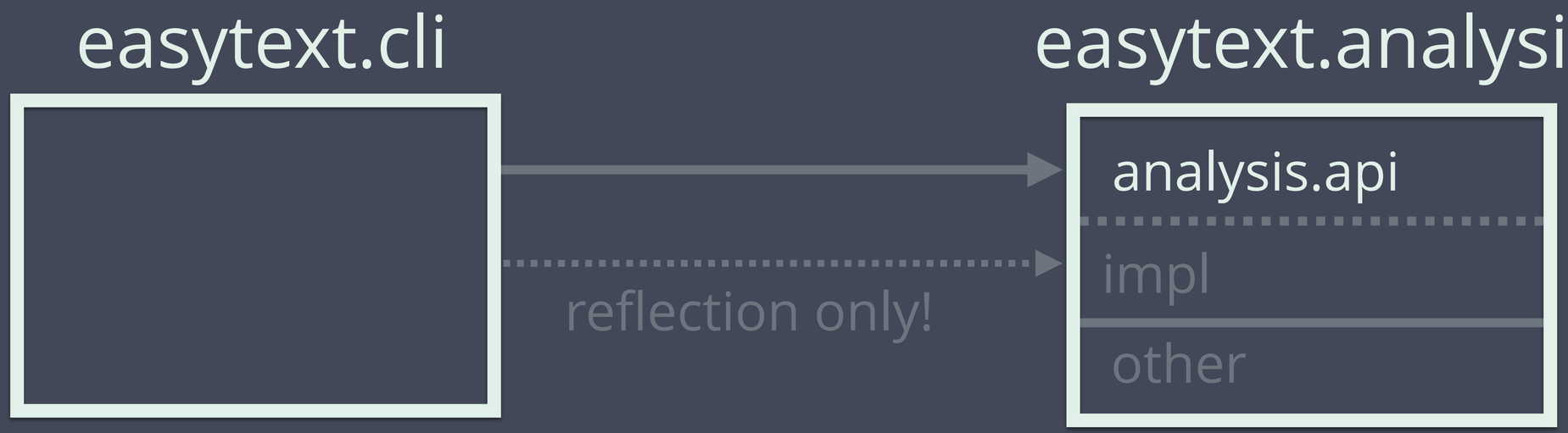


A Module Primer

```
module e
  require
}
```

Packages are encapsulated by default

```
module easytext.analysis {
  exports analysis.api;
  opens impl;
}
```

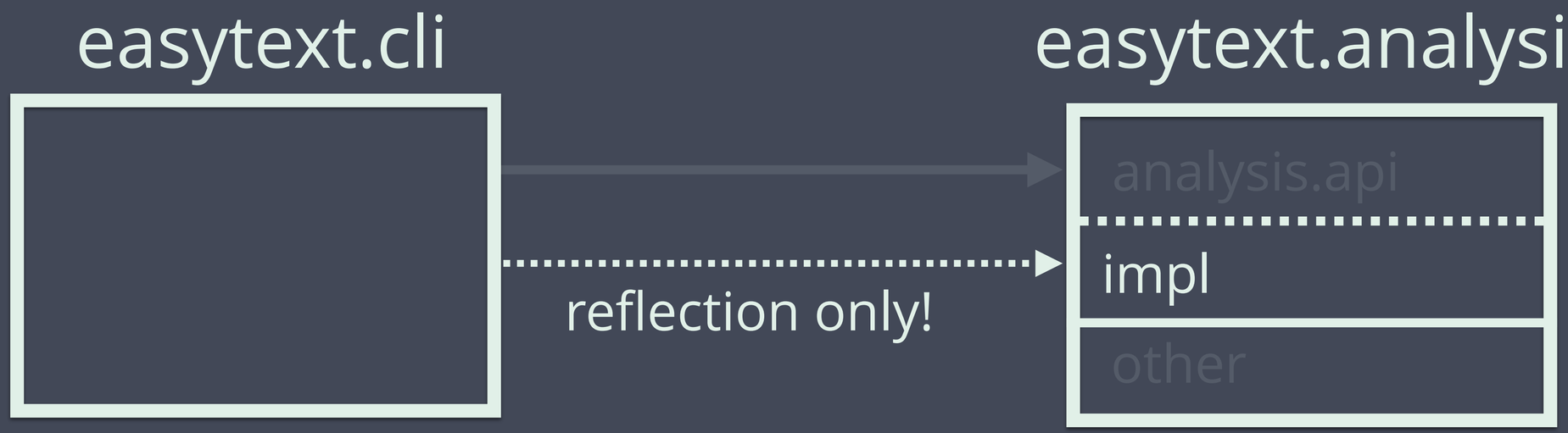


A Module Primer

```
module e
  require
}
```

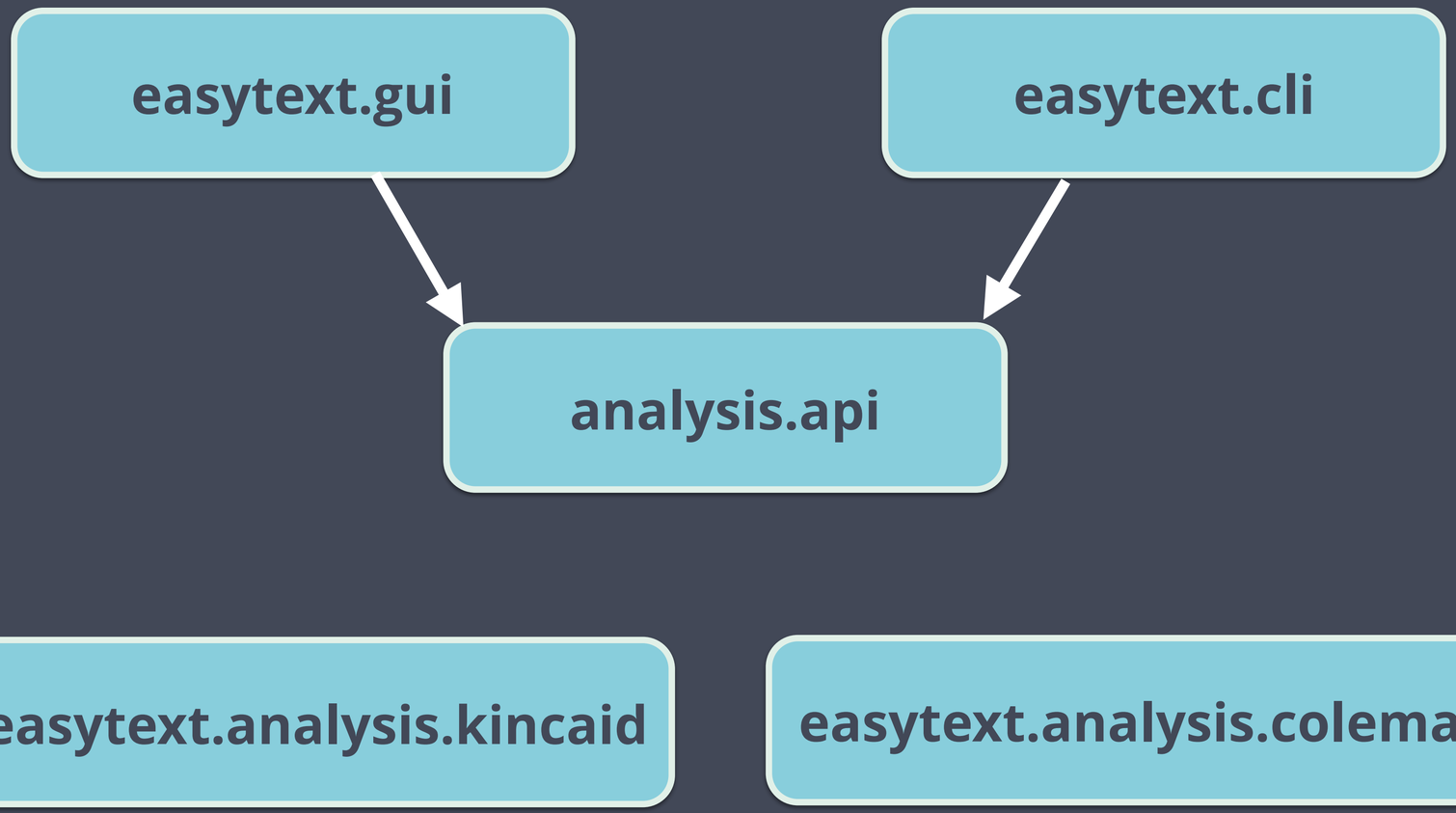
Packages can be “opened” for deep reflection at run-time

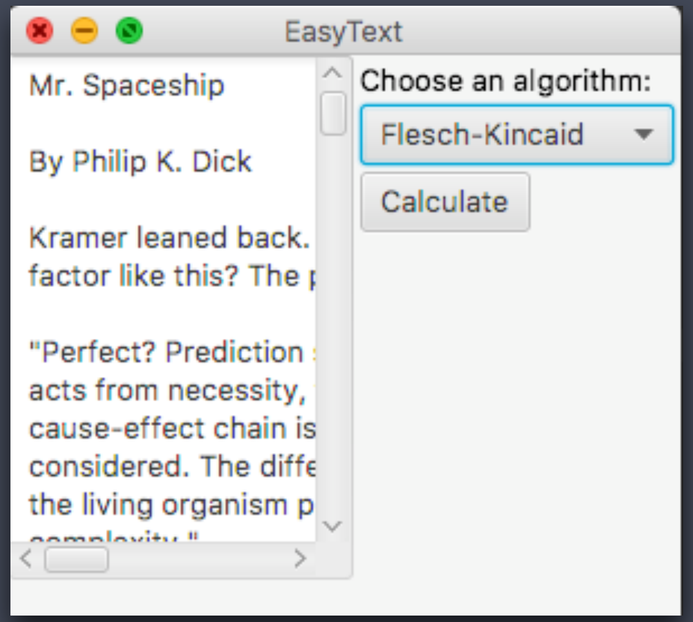
```
module easytext.analysis {
  exports analysis.api;
  opens impl;
}
```



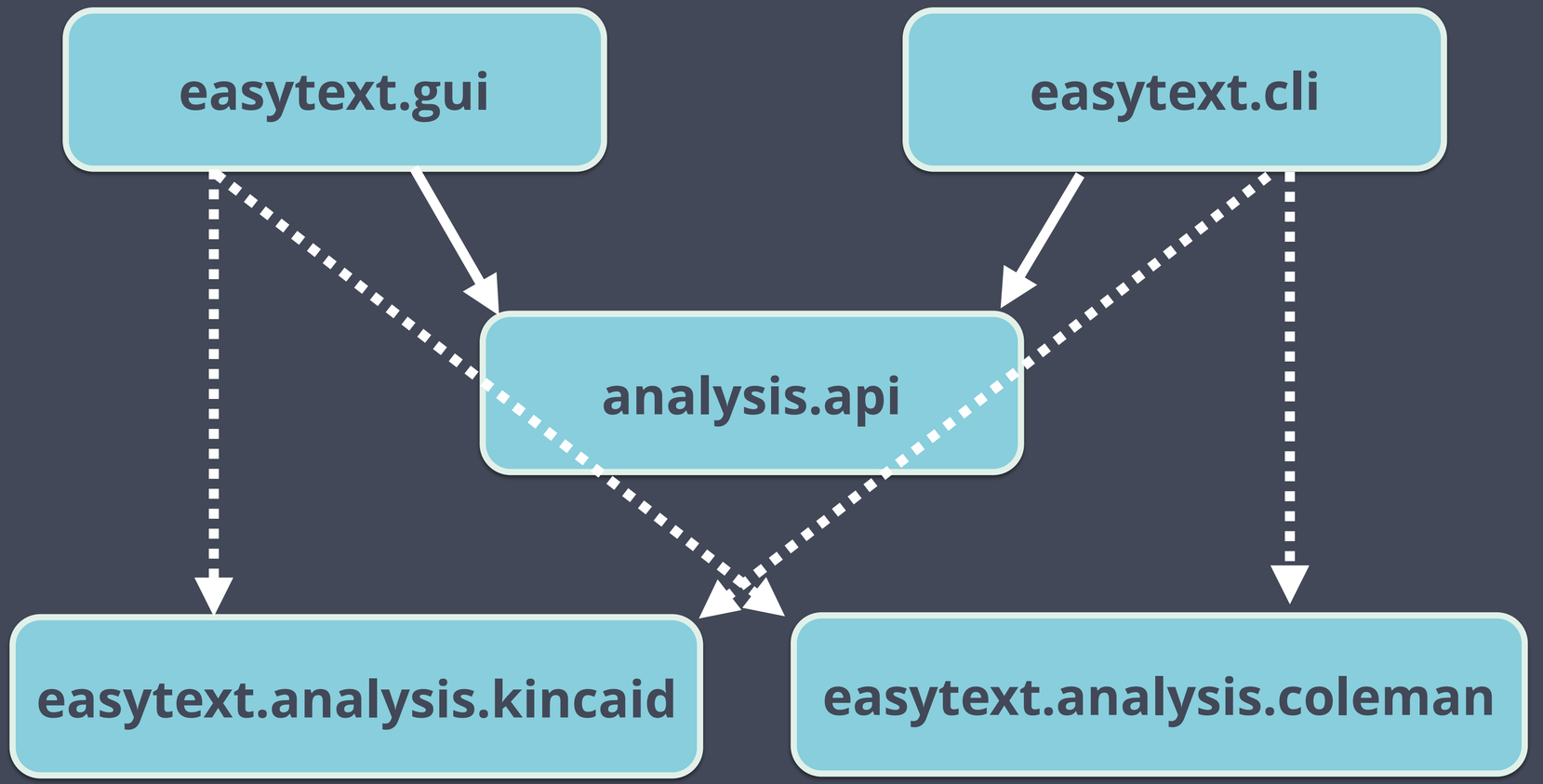
Services

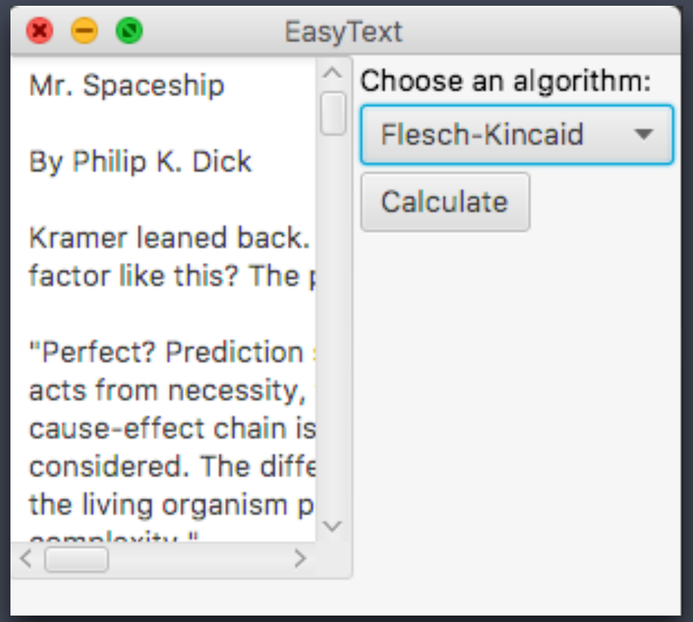
EasyText example



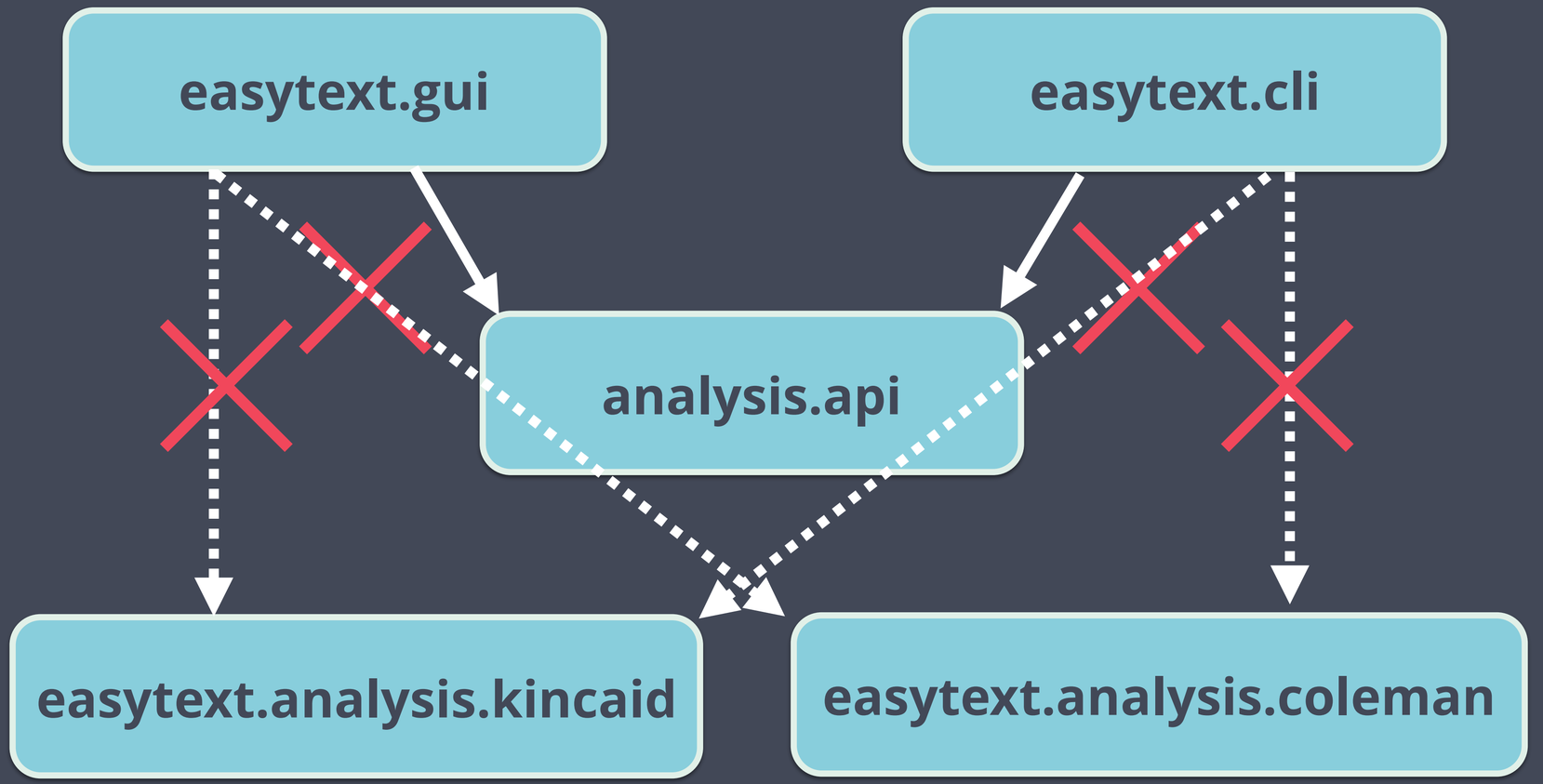


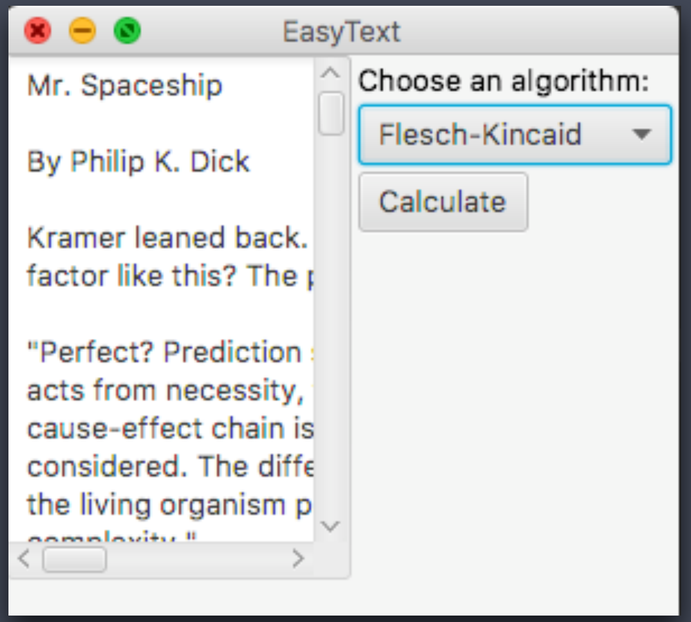
EasyText example





EasyText example





Encapsulation vs. decoupling

- Even with interfaces, an instance has to be created...

```
Analyzer i = new KincaidAnalyzer();
```

Encapsulation vs. decoupling

- Even with interfaces, an instance has to be created...

```
Analyzer i = new KincaidAnalyzer();
```

- We need to export our implementation! :-(

```
module easytext.analysis.kincaid {
    exports easytext.analysis.kincaid;
}
```

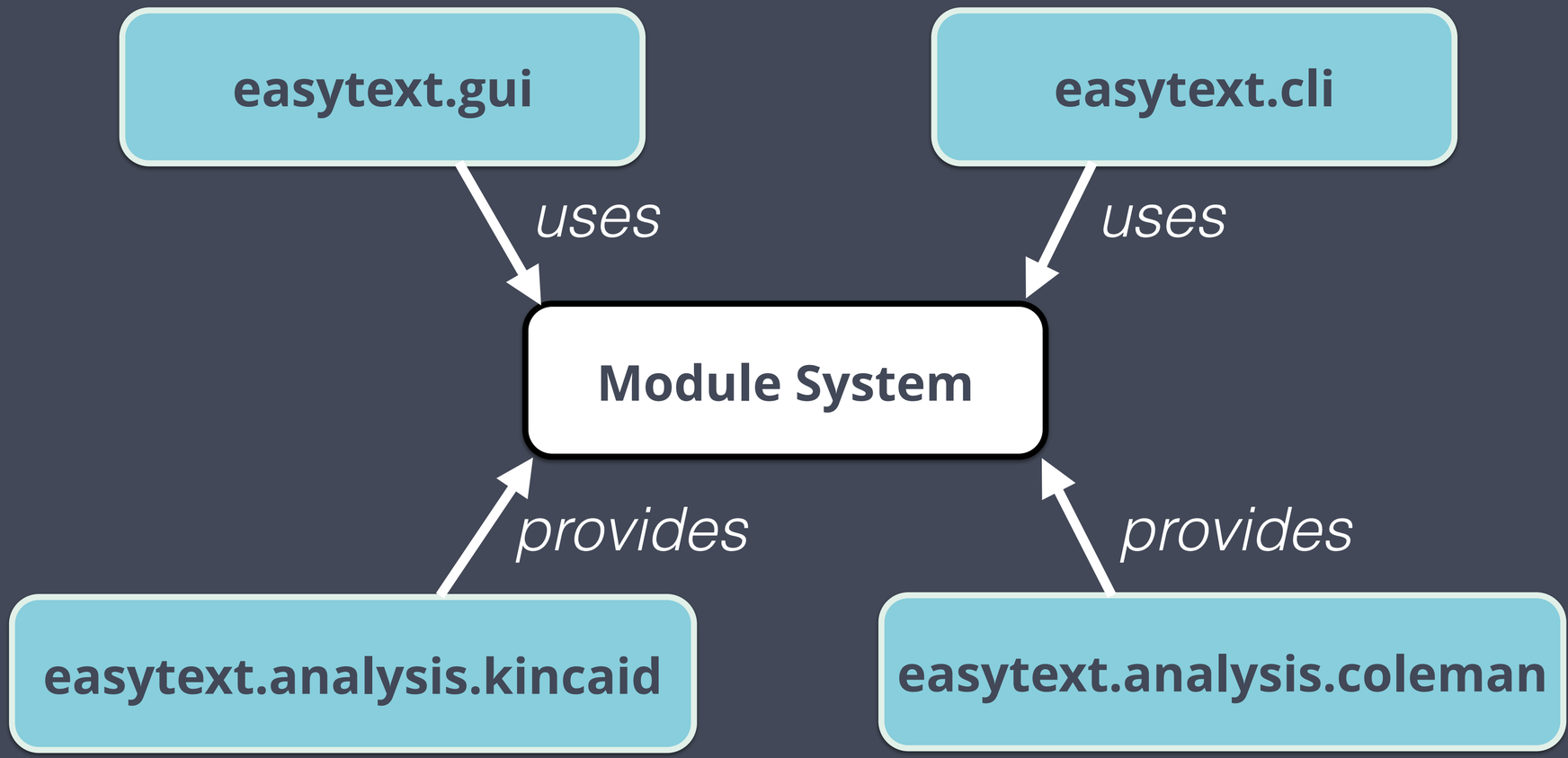
Reflection isn't a workaround

```
Class myImpl = Class.forName(…);
MyInterface i = myImpl.newInstance();
```

- Package needs to be open

```
module easytext.kincaid {
  opens easytext.analysis.kincaid;
}
```

Services to the rescue



Providing a service

```
module easytext.analysis.kincaid {
  requires easytext.analysis.api;

   provides javamodularity.easytext.analysis.api.Analyzer
       with javamodularity.easytext.analysis.kincaid.KincaidAnalyzer;

}
```

Implementing a service

- Just a plain Java class
- Not exported!

```
public class KincaidAnalyzer implements Analyzer {

    public KincaidAnalyzer() { }

    @Override
    public double analyze(List<List<String>> sentences) {
        …
    }
}
```

Consuming a service

```
module easytext.cli {
   requires easytext.analysis.api;

   uses javamodularity.easytext.analysis.api.Analyzer;
}
```

Consuming a service

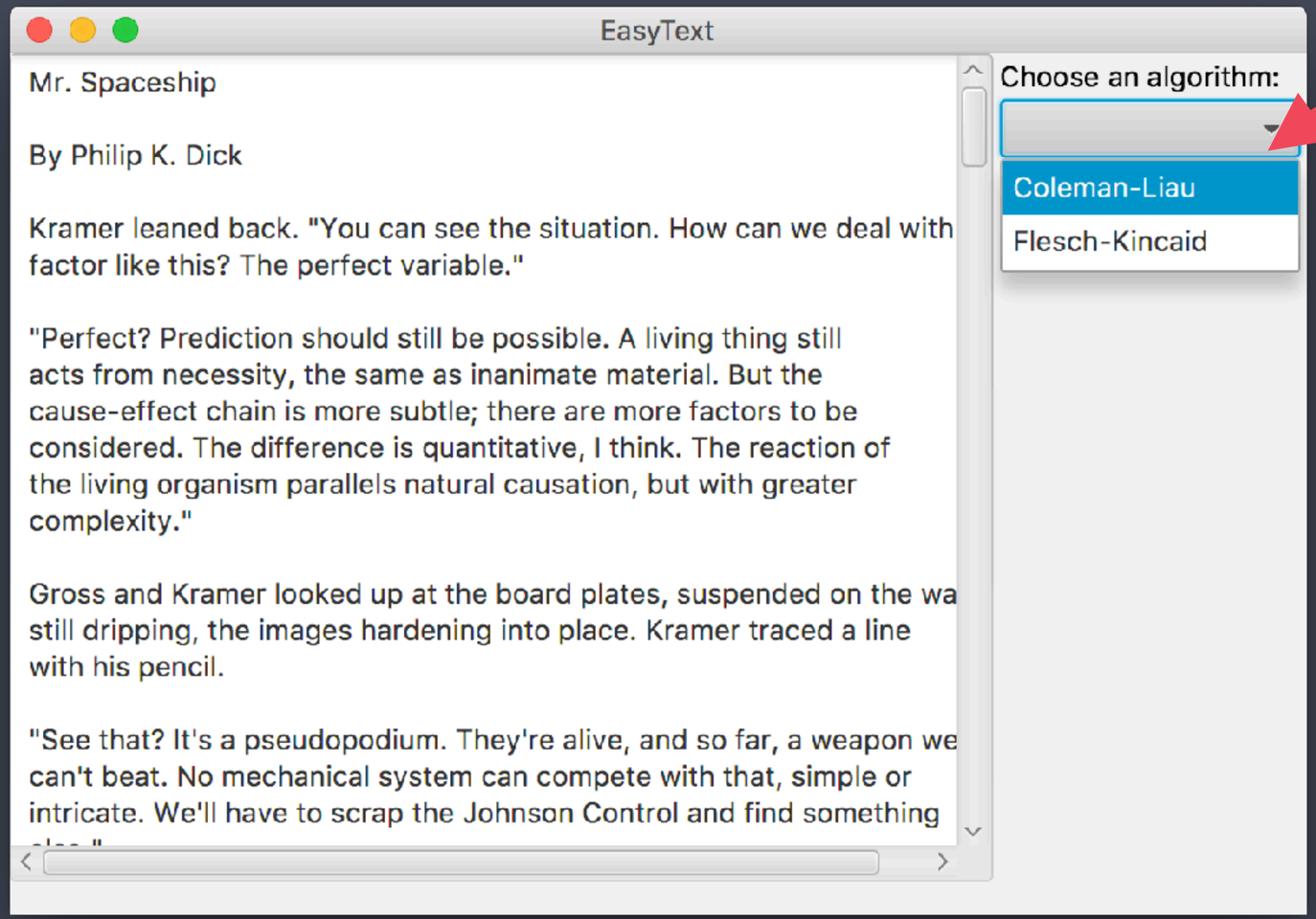
```
module easytext.cli {
  requires easytext.analysis.api;

  uses javamodularity.easytext.analysis.api.Analyzer;
}
```

```
Iterable<Analyzer> analyzers = ServiceLoader.load(Analyzer.class);

for (Analyzer analyzer: analyzers) {
  System.out.println(
    analyzer.getName() + ": " + analyzer.analyze(sentences));
}
```

Finding the right service

- ServiceLoader supports streams
- Lazy instantiation of services

```
ServiceLoader<Analyzer> analyzers =
   ServiceLoader.load(Analyzer.class);

analyzers.stream()
 .filter(provider -> …)
 .map(ServiceLoader.Provider::get)
 .forEach(analyzer -> System.out.println(analyzer.getName()));
```



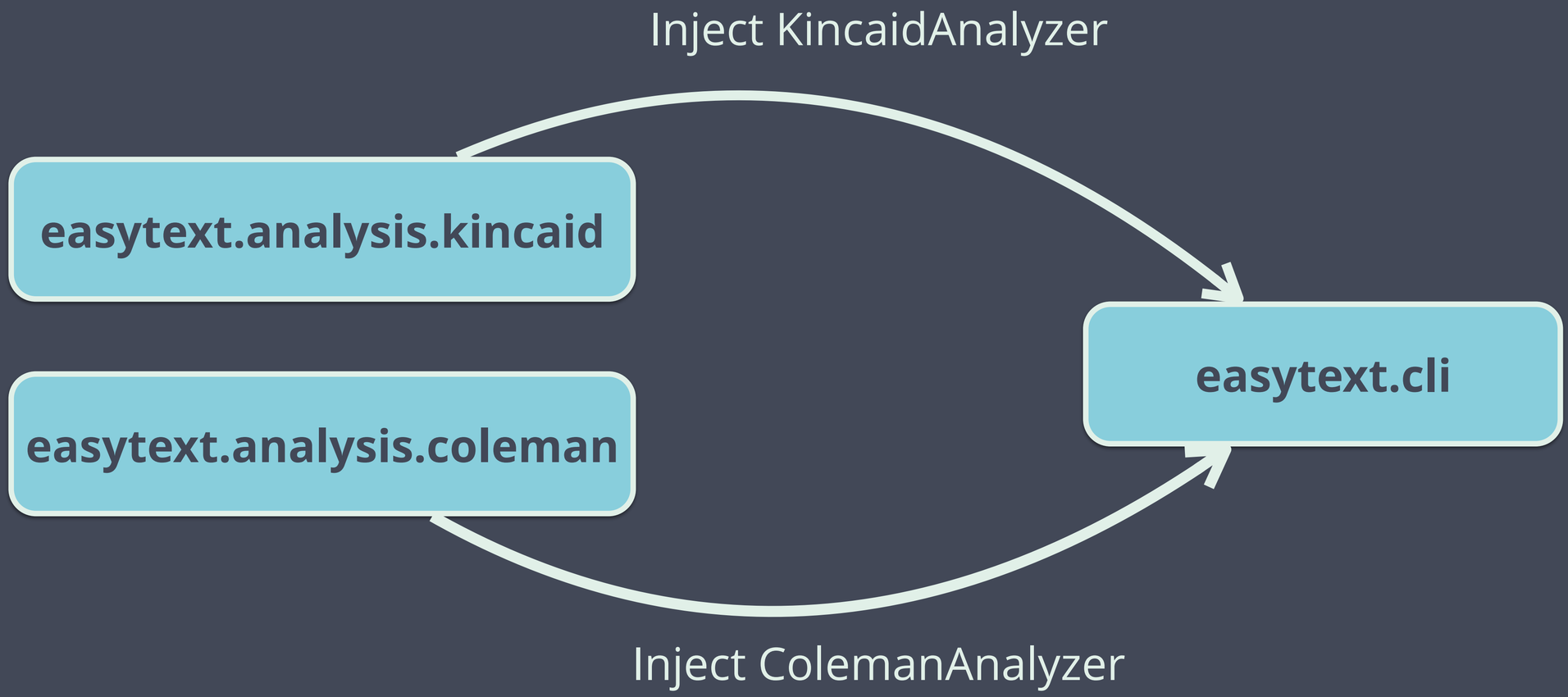
Add new analyses by adding provider modules on the module path

Dependency Injection

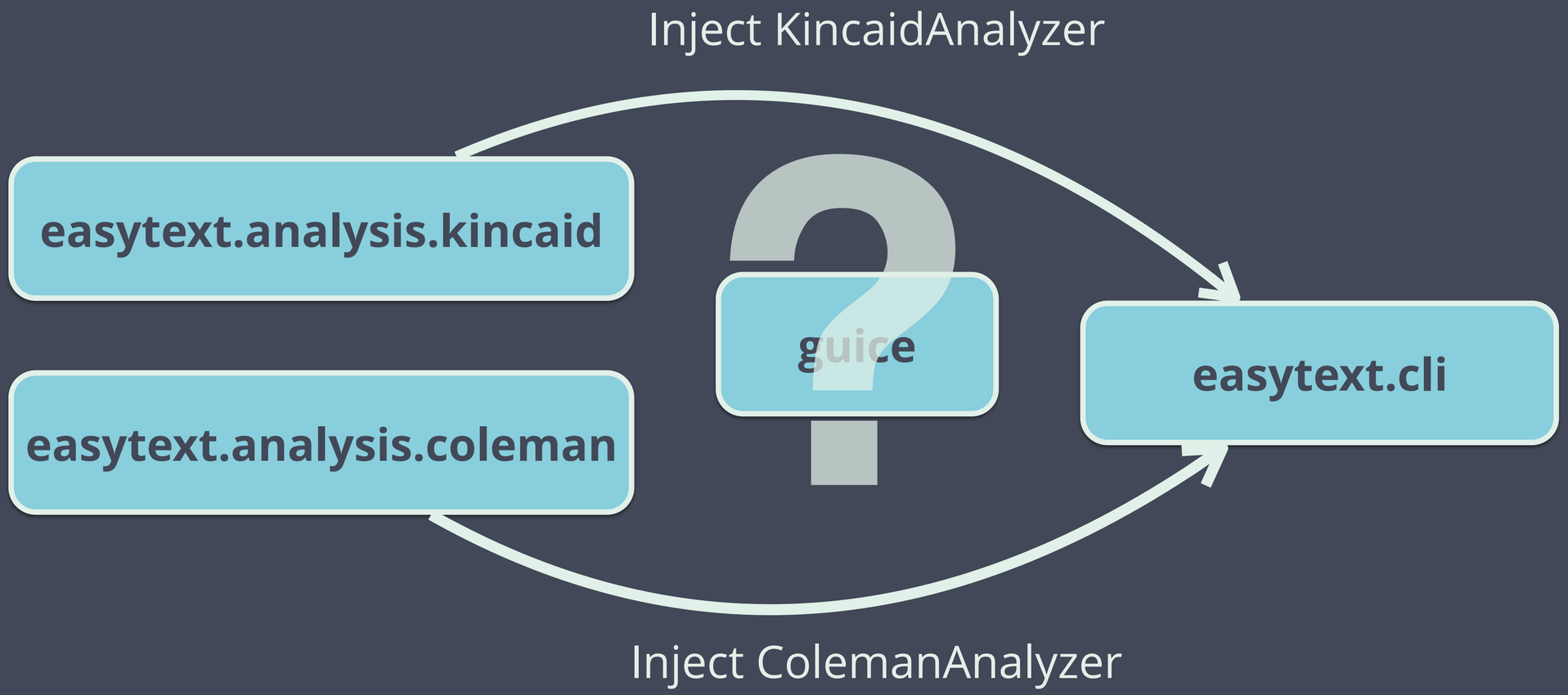
Dependency Injection

aka 'gimme back @Inject'

Goal: Inject Instances using Guice



Goal: Inject Instances using Guice



Providing a Guice service

- Provider module should export a Guice Module

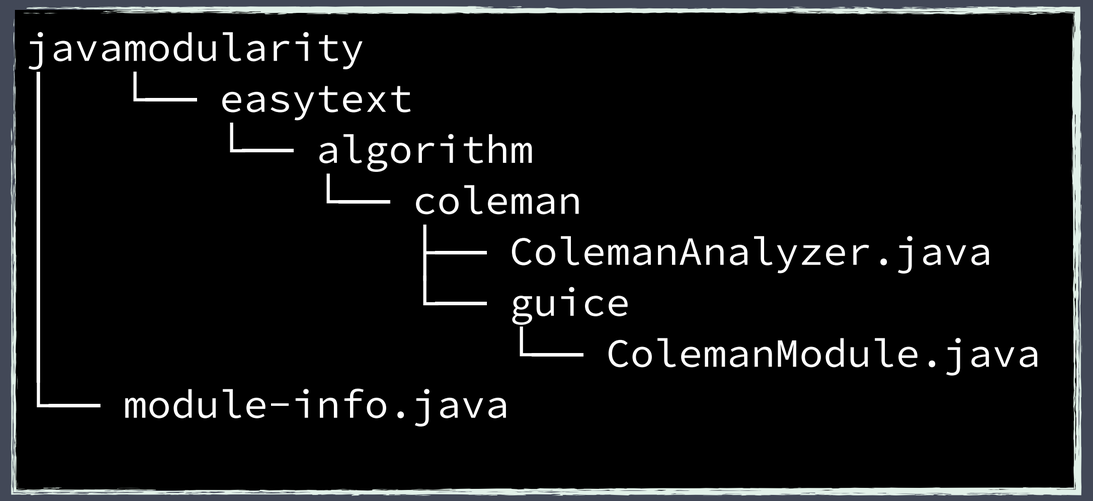
```
public class ColemanModule extends AbstractModule {

    @Override
    protected void configure() {
        Multibinder
            .newSetBinder(binder(), Analyzer.class)
            .addBinding().to(ColemanAnalyzer.class);
    }
}
```

Providing a Guice service

- Consumers must be able to compile against the Guice module
- Implementation package must be **open**

Providing a Guice service

- Consumers must be able to compile against the Guice module
- Implementation package must be **open**



Providing a Guice service

- Consumers must be able to compile against the Guice module
- Implementation package must be **open**

```
javamodularity
└── easytext
    └── algorithm
        └── coleman
            ├── ColemanAnalyzer.java
            └── guice
                └── ColemanModule.java
└── module-info.java
```

```
module easytext.algorithm.coleman {
   requires easytext.algorithm.api;
   requires guice;
   requires guice.multibindings;

   exports javamodularity.easytext.algorithm.coleman.guice;
   opens javamodularity.easytext.algorithm.coleman;
}
```

Consuming a Guice service

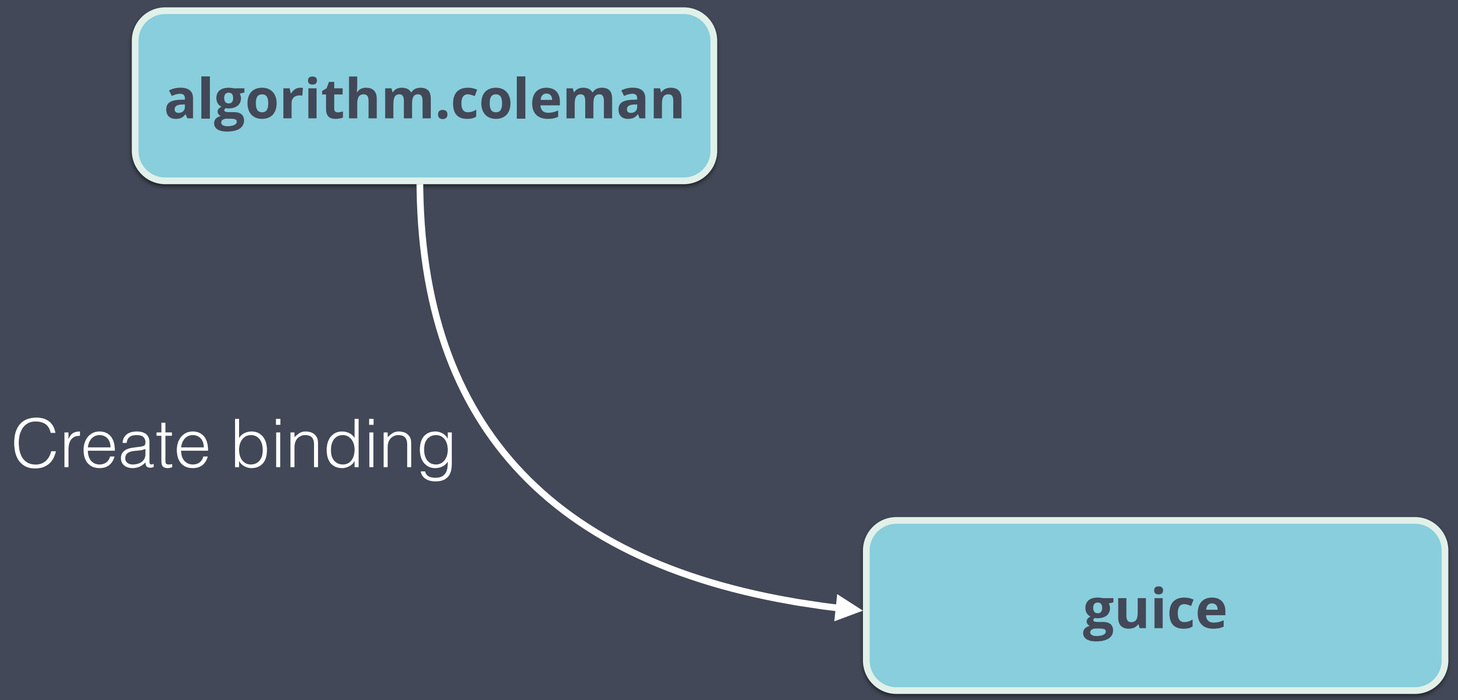
```
public class Main {

   public static void main(String... args) {
      Injector injector =
       Guice.createInjector(
          new ColemanModule(),
          new KincaidModule());

      CLI cli = injector.getInstance(CLI.class);
      cli.analyze(args[0]);
   }
}
```

Consuming a Guice service

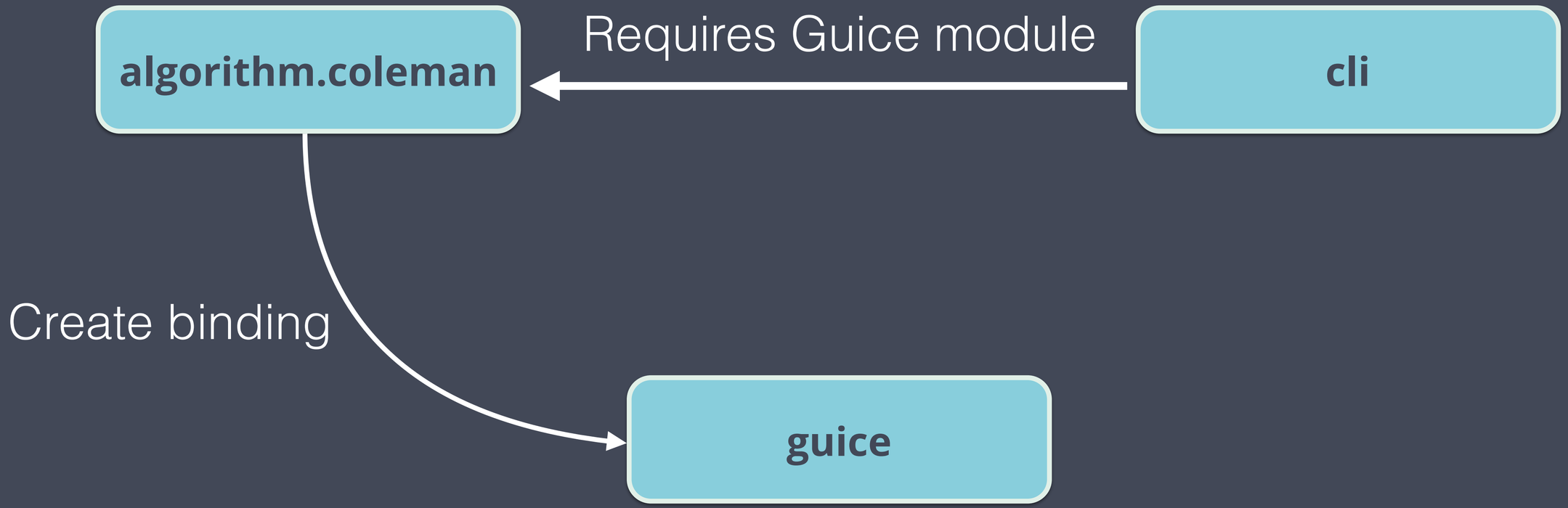
- Now we can @Inject

```
public class CLI {

    private final Set<Analyzer> analyzers;

    @Inject
    public CLI(Set<Analyzer> analyzers) {
        this.analyzers = analyzers;
    }

    …
```

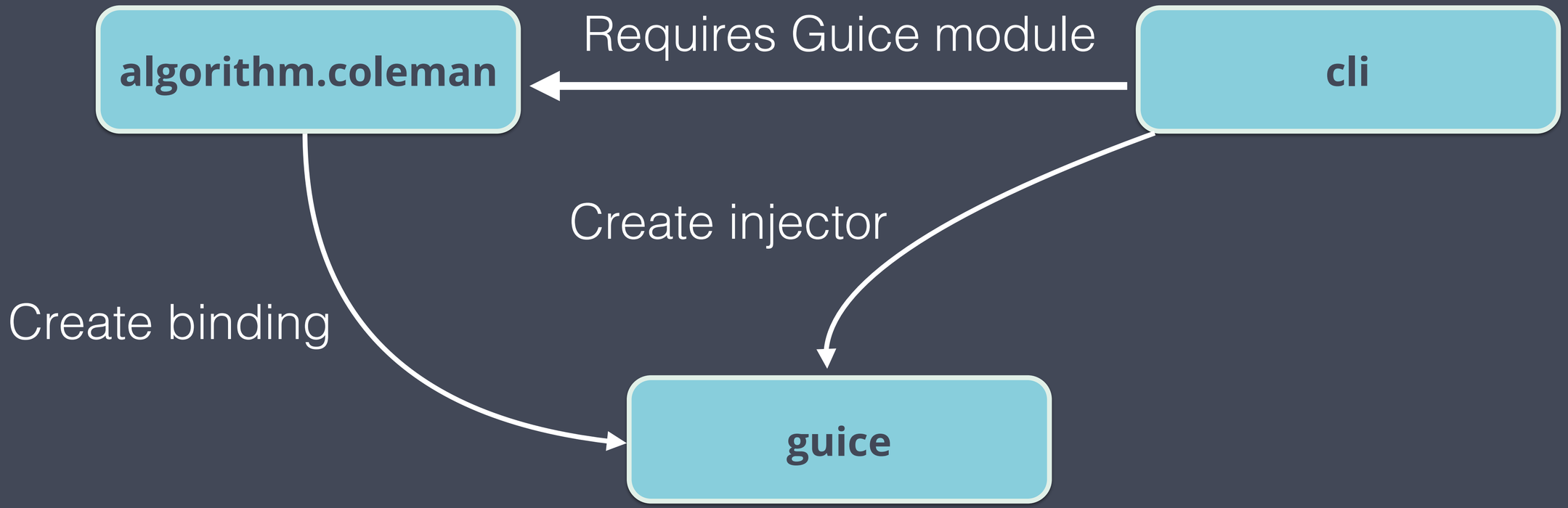
Injecting a service



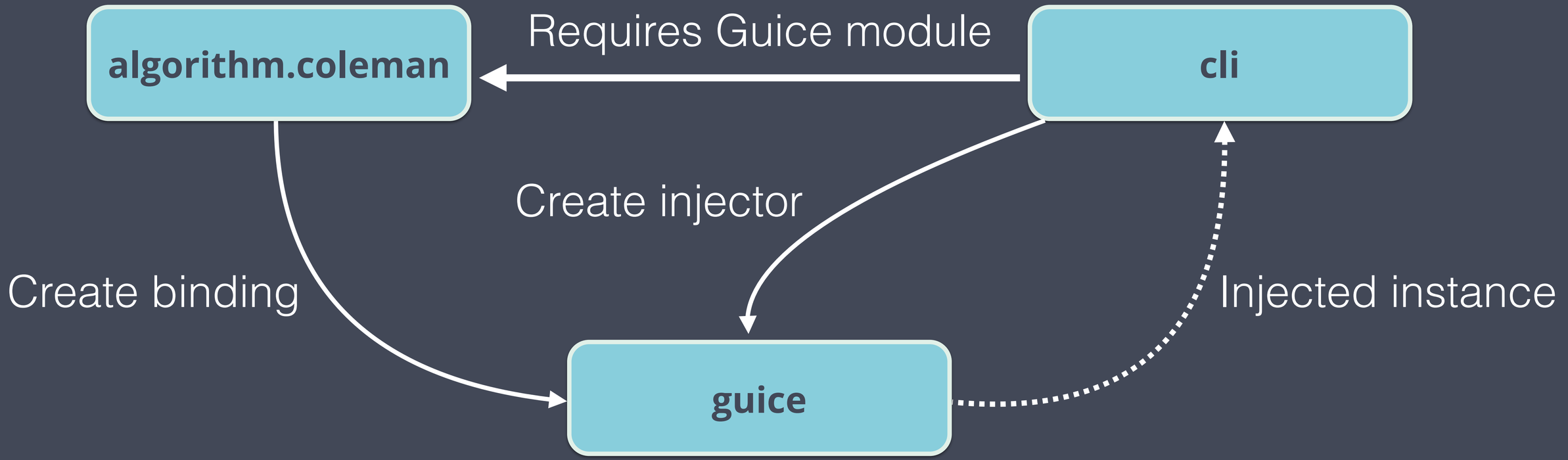
Injecting a service



Injecting a service



Injecting a service



Downsides of this setup

- Require the implementation Guice Modules
- Open package for Guice injection

```
module easytext.cli {
    requires easytext.algorithm.api;
    requires guice;
    requires easytext.algorithm.coleman;
    requires easytext.algorithm.kincaid;

    opens javamodularity.easytext.cli;
}
```

Services vs Guice

Benefits of using Guice

- @Inject vs using an API
- Developers might be more familiar with this model

Downsides of using Guice

- Requires more coupling
- Adding an implementation requires code changes
- Service binding not checked by module system

Services *and* Guice

```
module easytext.algorithm.coleman {
   requires easytext.algorithm.api;
   requires guice;
   requires guice.multibindings;

   exports javamodularity.easytext.algorithm.coleman.guice;
   opens javamodularity.easytext.algorithm.coleman;
}
```

Services *and* Guice

```
module easytext.algorithm.coleman {
   requires easytext.algorithm.api;
   requires guice;
   requires guice.multibindings;

   exports javamodularity.easytext.algorithm.coleman.guice;
   opens javamodularity.easytext.algorithm.coleman;
}
```

```
provides com.google.inject.AbstractModule
    with javamodularity.easytext.algorithm.coleman.guice.ColemanModule
```

Services *and* Guice

```
module easytext.cli {
    requires easytext.algorithm.api;
    requires guice;
    requires easytext.algorithm.coleman;
    requires easytext.algorithm.kincaid;

    opens javamodularity.easytext.cli;
}
```

Services *and* Guice

```
module easytext.cli {
    requires easytext.algorithm.api;
    requires guice;
    requires easytext.algorithm.coleman;
    requires easytext.algorithm.kincaid;

    opens javamodularity.easytext.cli;
}
```

uses com.google.inject.AbstractModule

Services *and* Guice

```
public static void main(String... args) throws IOException {
  Injector injector = Guice.createInjector(
          ServiceLoader.load(AbstractModule.class));

  CLI cli = injector.getInstance(CLI.class);
  cli.analyze(args[0]);
}
```

Modular Design

Naming Modules

... is hard

Naming Modules

... is hard

Application modules

- Short & memorable
- <application>.<component>
- e.g. **easytext.gui**

Naming Modules

... is hard

Application modules vs. Library modules

- Short & memorable
- <application>.<component>
- e.g. **easytext.gui**

- Uniqueness
- Reverse DNS
- **com.mycompany.ourlib**
- 'Root package'

API Modules

API Modules

- Module with exported API only
  - When multiple implementations are expected
  - Can also contain 'default' implementation

API Modules

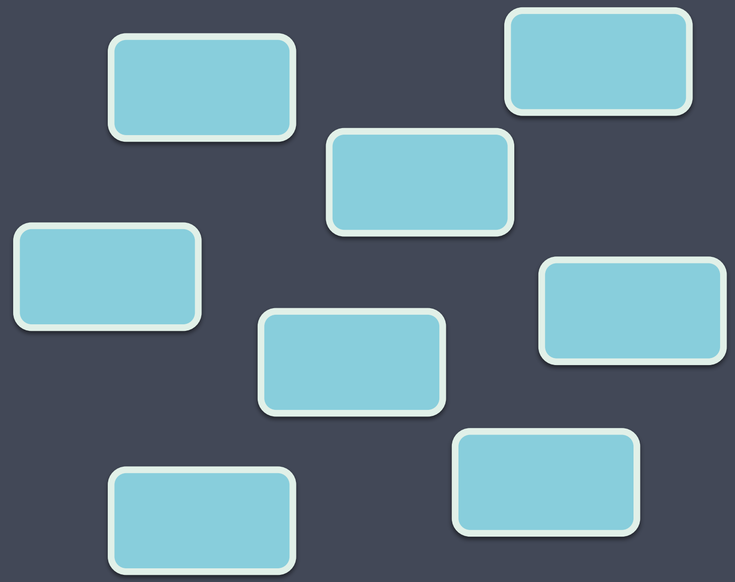
- Module with exported API only
  - When multiple implementations are expected
  - Can also contain 'default' implementation
- API modules export:
  - Interfaces
  - (Abstract) classes
  - Exceptions
  - Etc.

Small Modules vs. All-in-one

Small Modules vs. All-in-one

Small modules

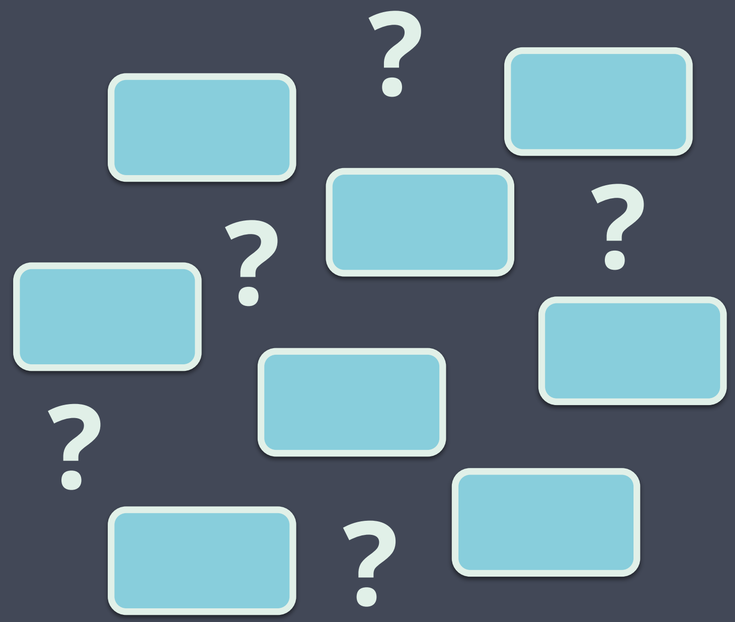
- Composability
- Reusability



Small Modules vs. All-in-one

Small modules

- Composability
- Reusability



All-in-one modules

- Ease-of-use

Small Modules vs. All-in-one

Why choose? Use aggregator modules

Small Modules vs. All-in-one

Why choose? Use aggregator modules

Module without code, using implied readability:

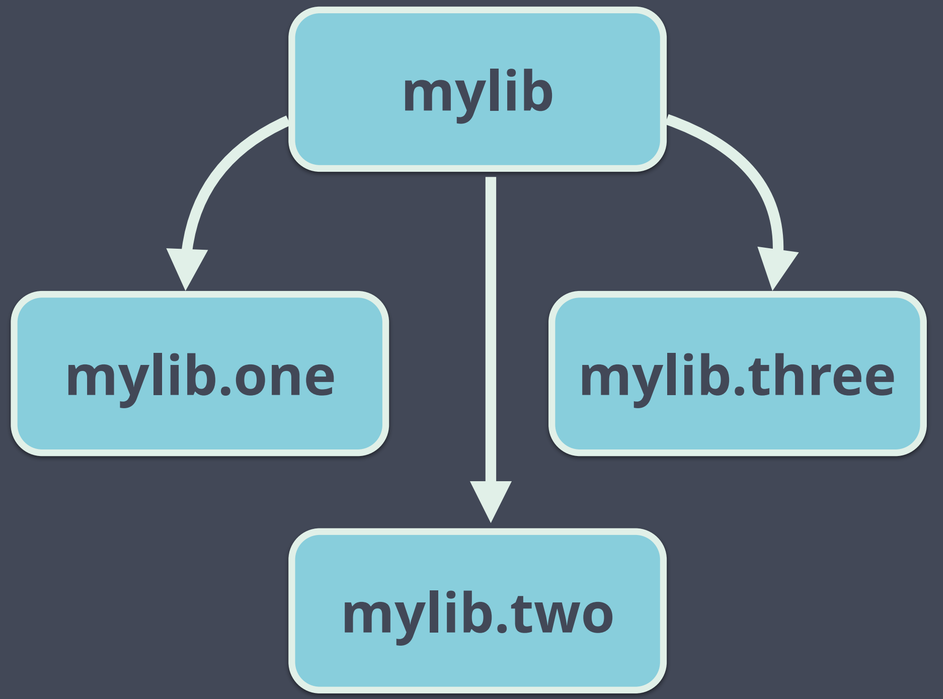
```
module mylib {
      requires transitive mylib.one;
      requires transitive mylib.two;
      requires transitive mylib.three;
}
```

Small Modules vs. All-in-one

Why choose? Use aggregator modules

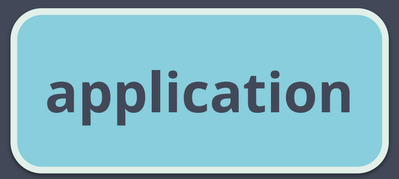
Module without code, using implied readability:

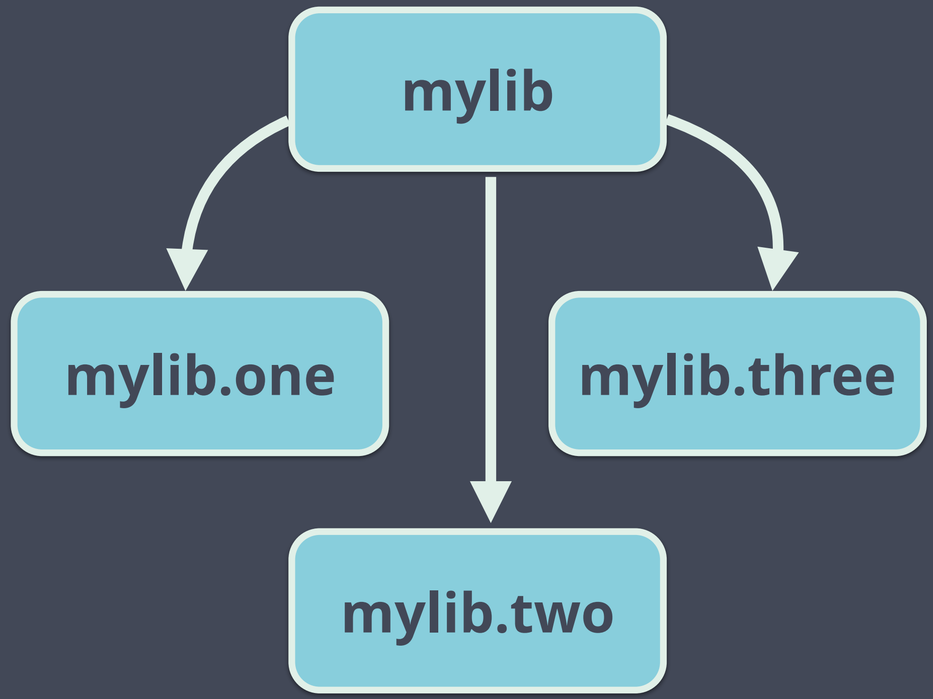
```
module mylib {
      requires transitive mylib.one;
      requires transitive mylib.two;
      requires transitive mylib.three;
}
```



Small Modules vs. All-in-one

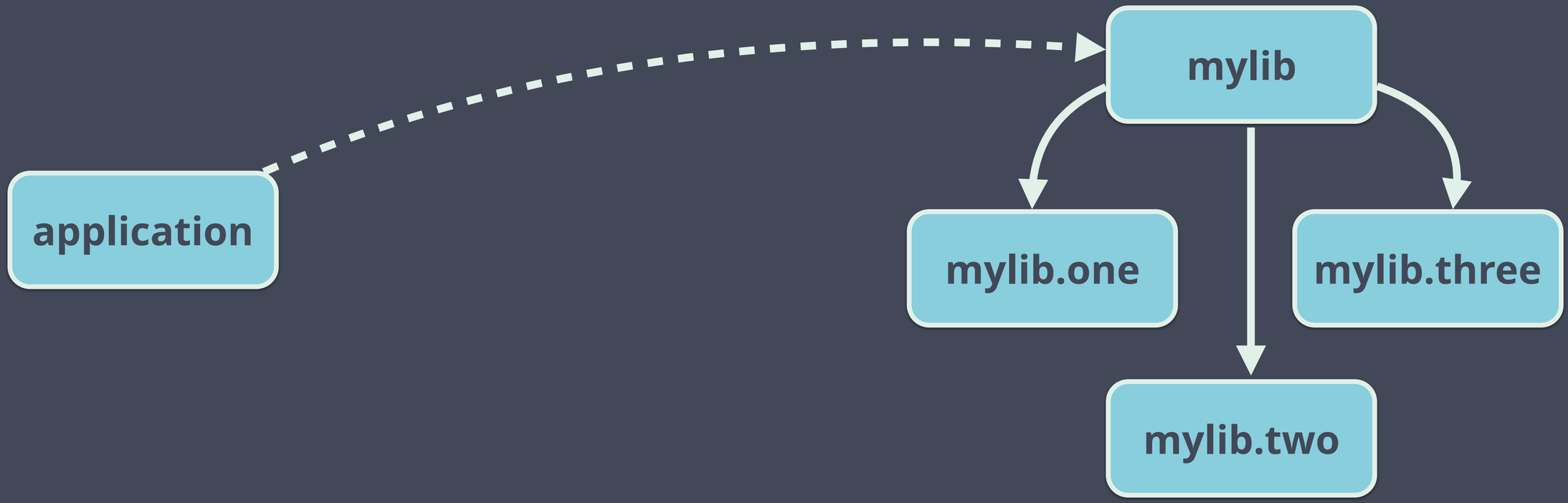
Why choose? Use aggregator modules





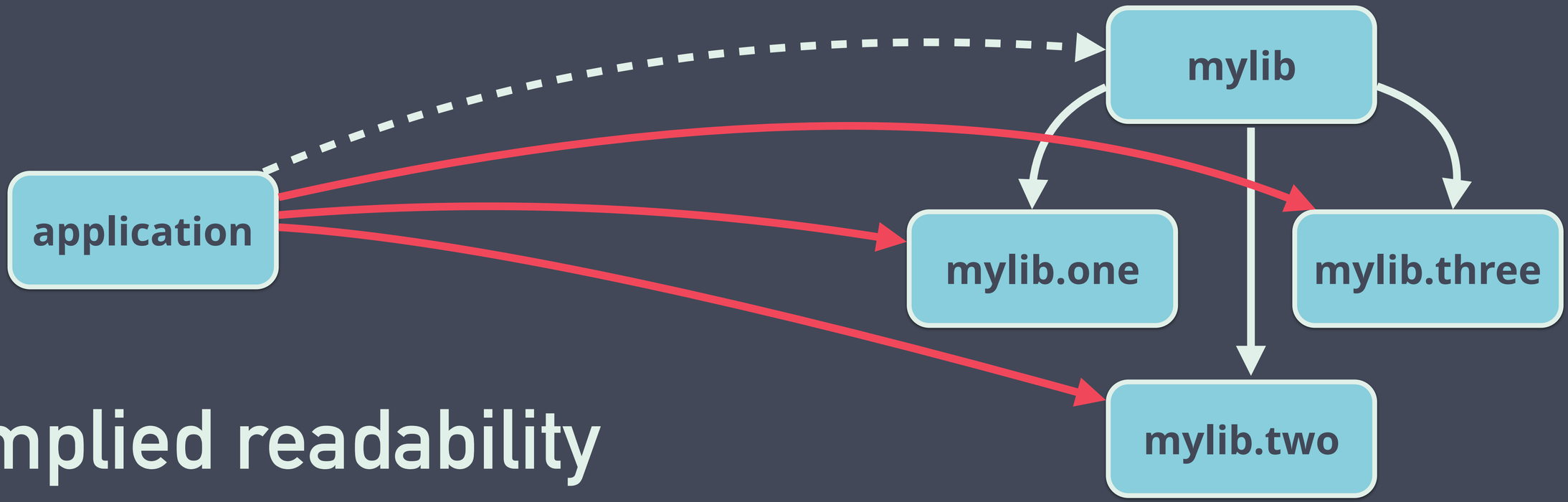
Small Modules vs. All-in-one

Why choose? Use aggregator modules



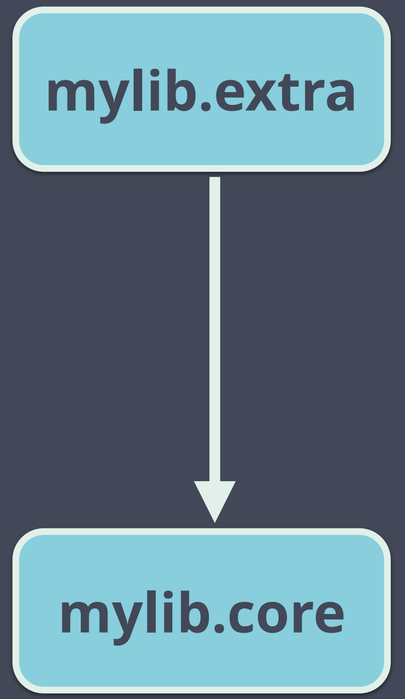
Small Modules vs. All-in-one

Why choose? Use aggregator modules

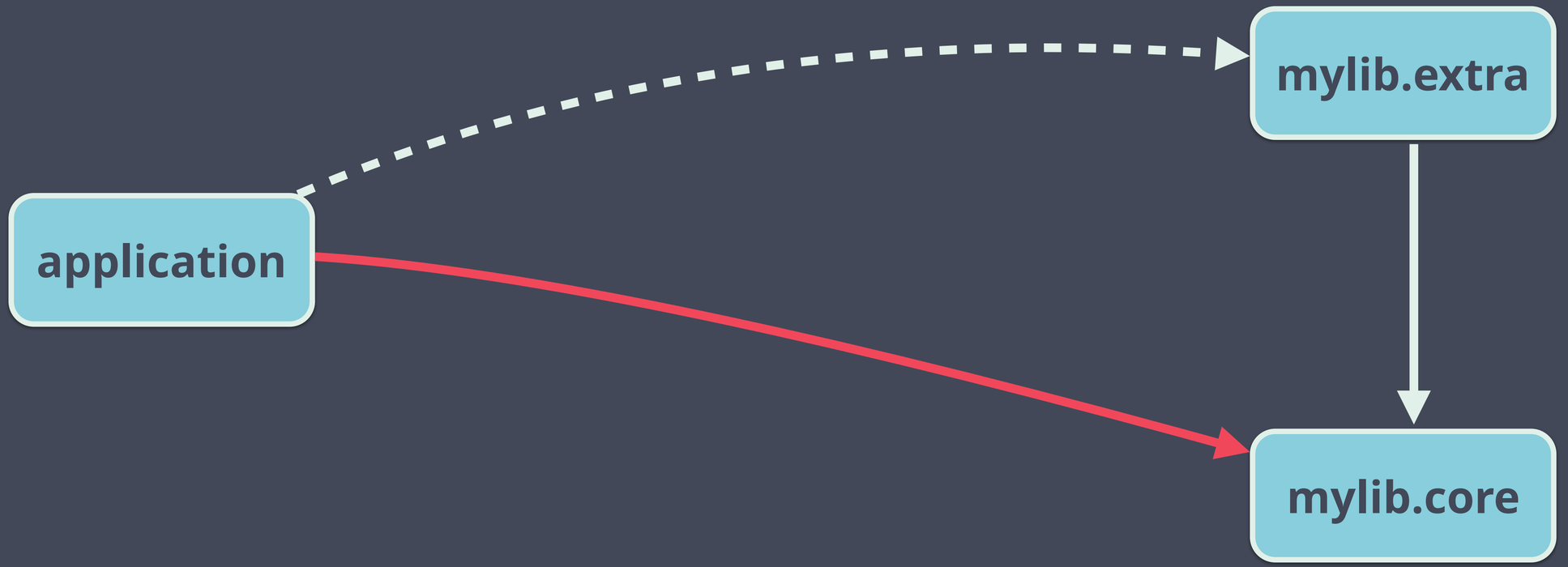


Implied readability

Small Modules vs. All-in-one

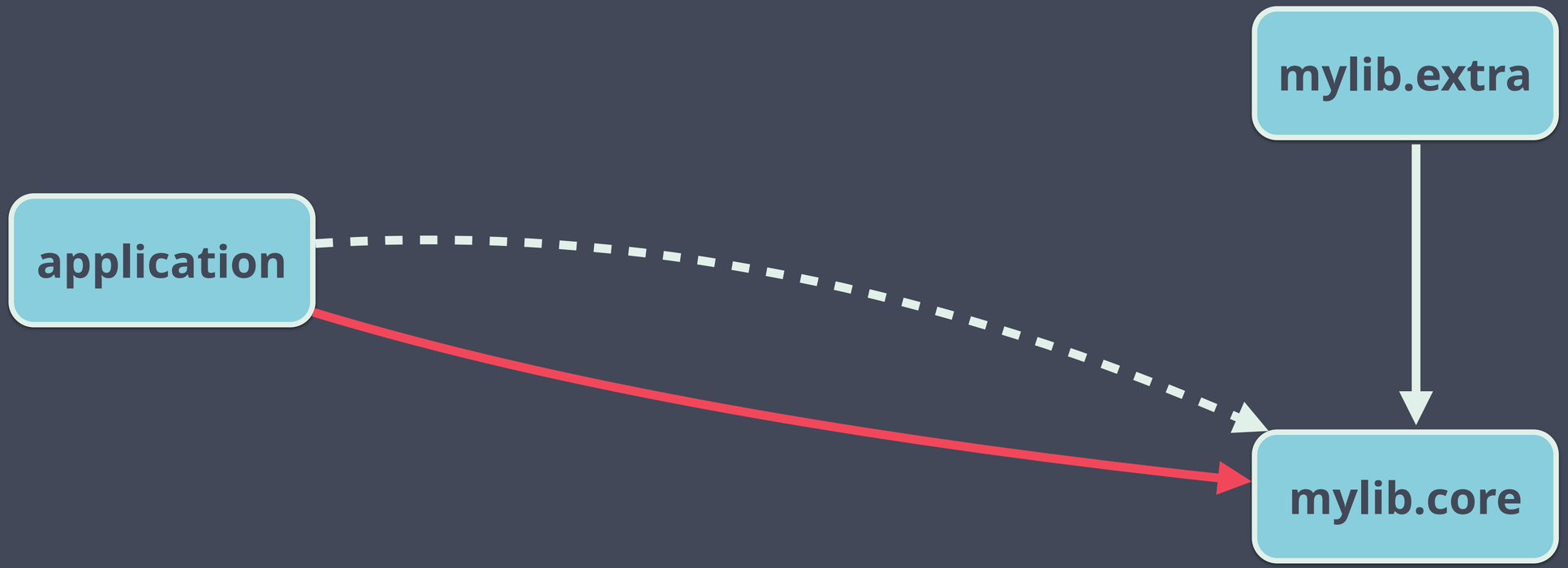


Small Modules vs. All-in-one



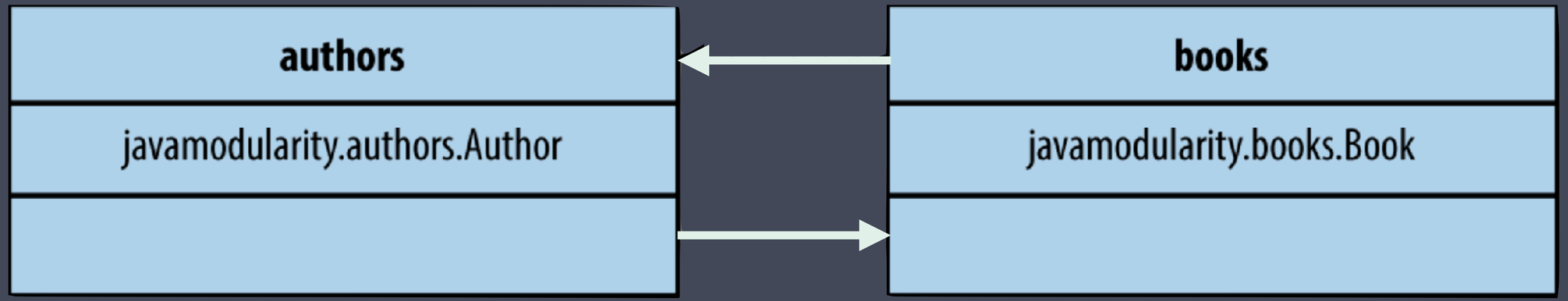
Small Modules vs. All-in-one

Small Modules vs. All-in-one

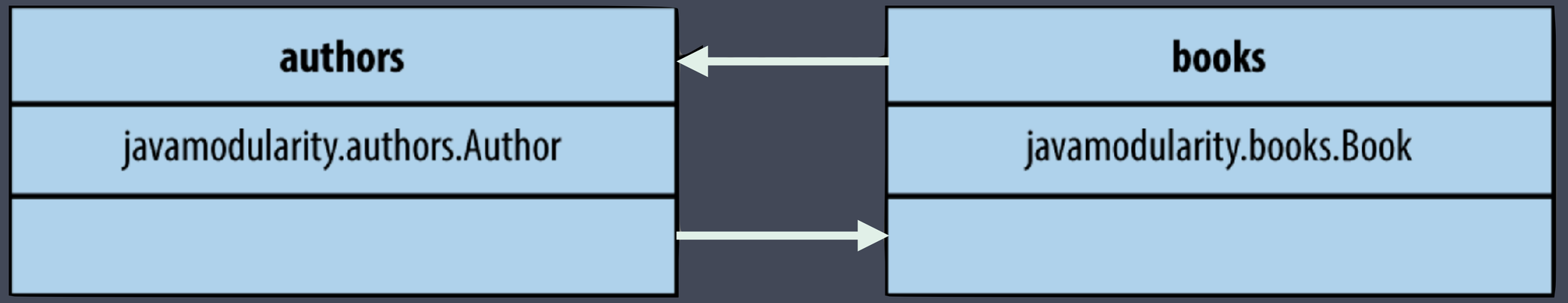


Breaking cycles

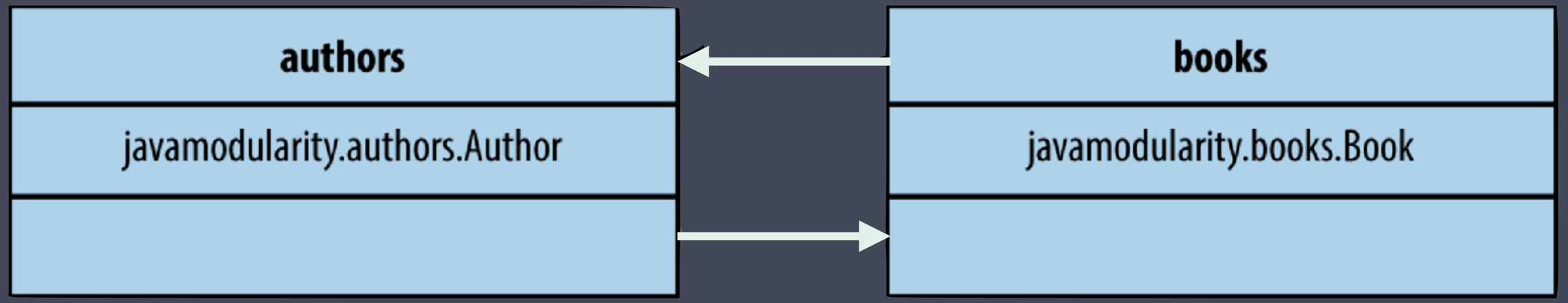
- Non-modular JARs can have cyclic dependencies
- Modules **can not**



Breaking cycles

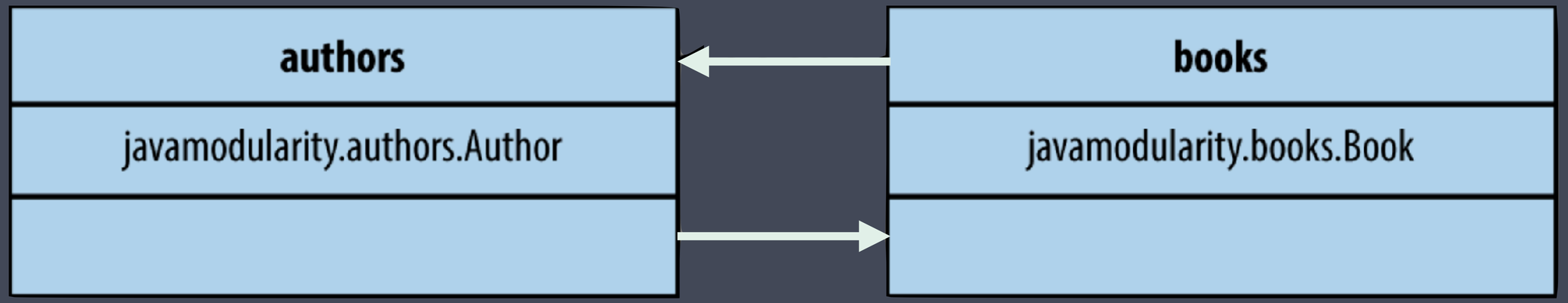


Breaking cycles




```
public class Author {
  private String name;
  private List<Book> books;

  public Author(String name)
  {
    this.name = name;
  }
  // ..
}
```

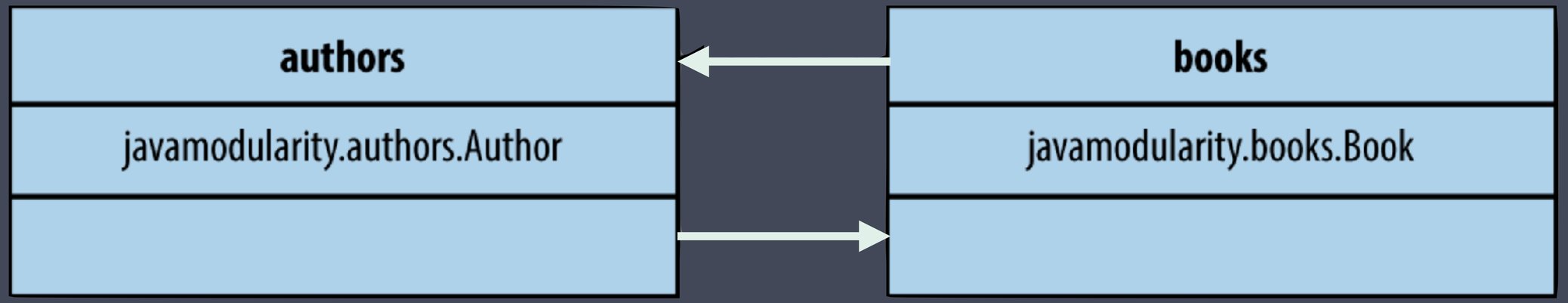
Breaking cycles




```
public class Author {
  private String name;
  private List<Book> books;

  public Author(String name)
  {
    this.name = name;
  }
  // ..
}
```

```
public class Book {
  public Author author;
  public String title;

  public void printBook() {
    System.out.printf(
      "%s, by %s\n\n%s",
      title, author.getName(),
      text);
  }
```

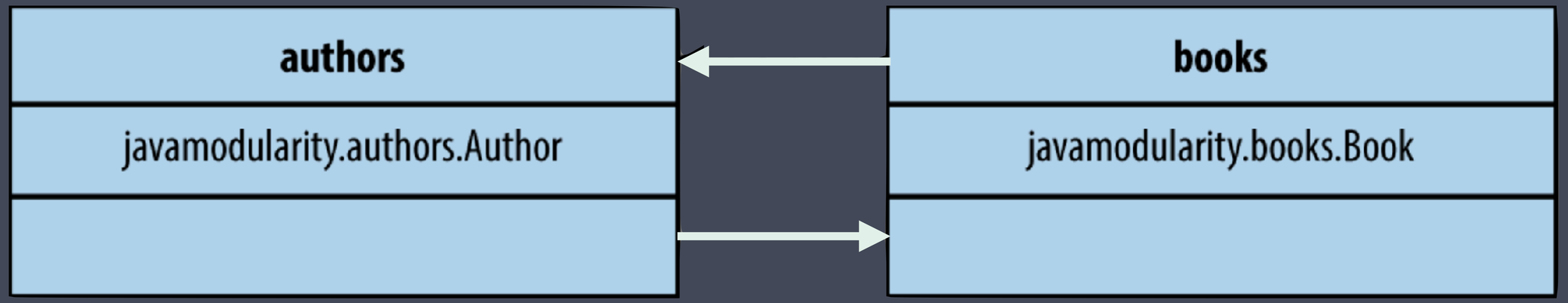
Breaking cycles




```
public class Author {
  private String name;
  private List<Book> books;

  public Author(String name)
  {
    this.name = name;
  }
  // ..
}
```

```
public class Book {
  public Author author;
  public String title;

  public void printBook() {
    System.out.printf(
      "%s, by %s\n\n%s",
      title, author.getName(),
      text);
  }
```

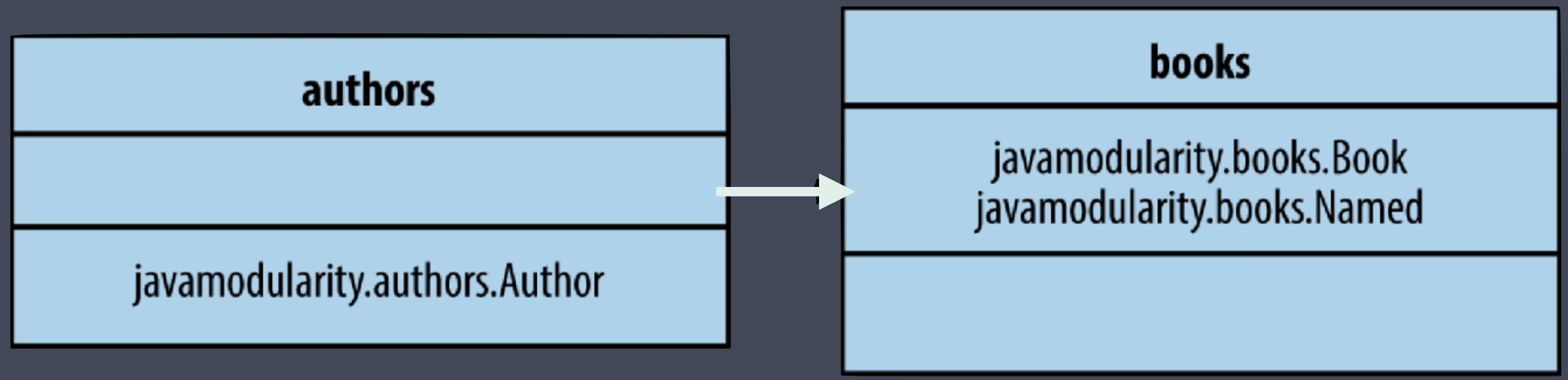
Breaking cycles



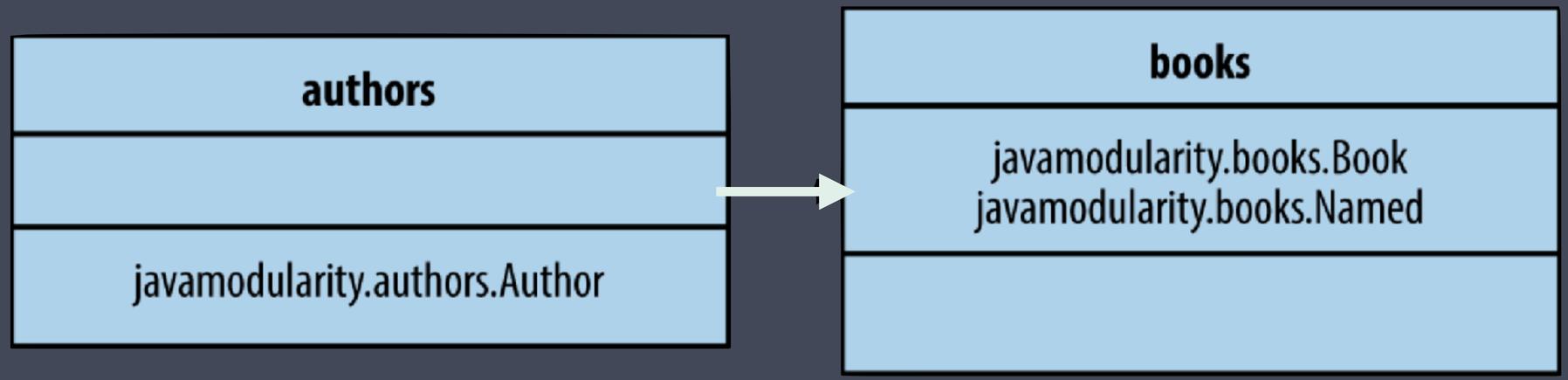

```
public class Author {
  private String name;
  private List<Book> books;

  public Author(String name)
  {
    this.name = name;
  }
  // ..
}
```

```
public class Book {
  public Author author;
  public String title;

  public void printBook() {
    System.out.printf(
      "%s, by %s\n\n%s",
      title, author.getName(),
      text);
  }
```

Breaking cycles: abstraction

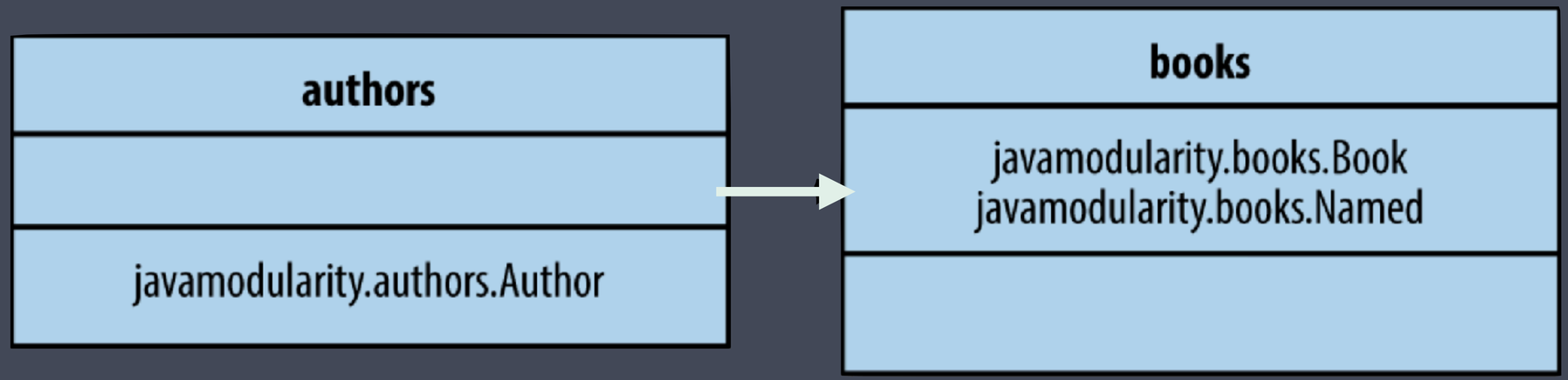


Breaking cycles: abstraction




```
public class Author
          implements Named {
  // ..

  public String getName() {
    return this.name;
  }

}
```

Breaking cycles: abstraction




```
public class Author
          implements Named {
   // ..

   public String getName() {
      return this.name;
   }

}
```

```
public interface Named {

   String getName();

}
```

Optional dependencies

- Requires means compile-time **and** run-time dependency
- What if we want an optional dependency?
  - Use it if available at run-time
  - Otherwise, run without

Optional dependencies

- Requires means compile-time **and** run-time dependency
- What if we want an optional dependency?
  - Use it if available at run-time
  - Otherwise, run without

Compile-time only dependencies:

```
module framework {
  requires static fastjsonlib;
}
```

Optional dependencies

- Requires means compile-time **and** run-time dependency
- What if we want an optional dependency?
  - Use it if available at run-time
  - Otherwise, run without

Compile-time only dependencies:

```
module framework {
  requires static fastjsonlib;
}
```

Resolve `fastjsonlib` if available at run-time, ignore if not

Optional dependencies

Avoid run-time exceptions!

Optional dependencies

Avoid run-time exceptions!

```
try {
   Class<?> clazz =
      Class.forName("javamodularity.fastjsonlib.FastJson");
   FastJson instance =
      (FastJson) clazz.getConstructor().newInstance();
    System.out.println("Using FastJson");
} catch (ReflectiveOperationException e) {
   System.out.println("Oops, we need a fallback!");
}
```

Optional dependencies

Better yet: use services for optional dependencies

```
module framework {
  requires json.api;

  uses json.api.Json;
}
```

```
ServiceLoader.load(Json.class)
   .stream()
   .filter(this::isFast)
   .findFirst();
```

Layers & Dynamic Module Loading

ModuleLayer

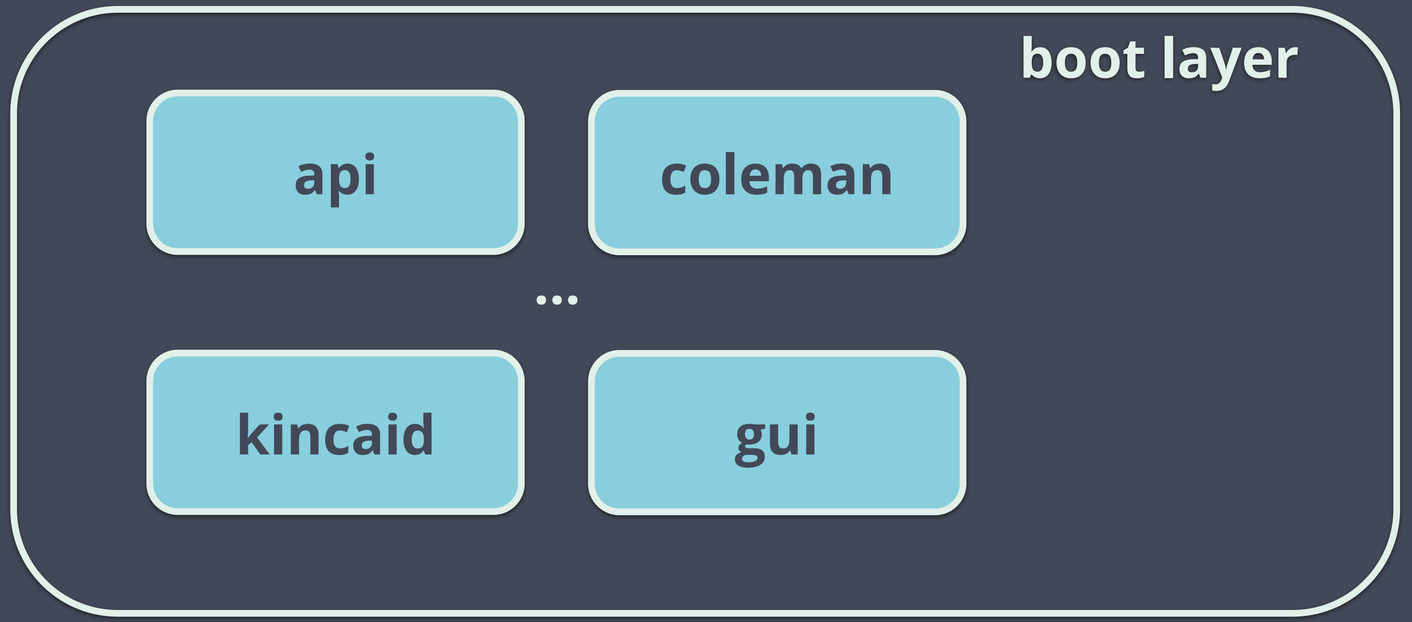
Module path

```
mods
├── easytext.algorithm.api
├── easytext.algorithm.coleman
├── easytext.algorithm.kincaid
├── easytext.algorithm.naivesyllablecounter
├── easytext.algorithm.nextgensyllablecounter
├── easytext.cli
└── easytext.gui
```

ModuleLayer

Module path

```
mods
├── easytext.algorithm.api
├── easytext.algorithm.coleman
├── easytext.algorithm.kincaid
├── easytext.algorithm.naivesyllablecounter
├── easytext.algorithm.nextgensyllablecounter
├── easytext.cli
└── easytext.gui
```



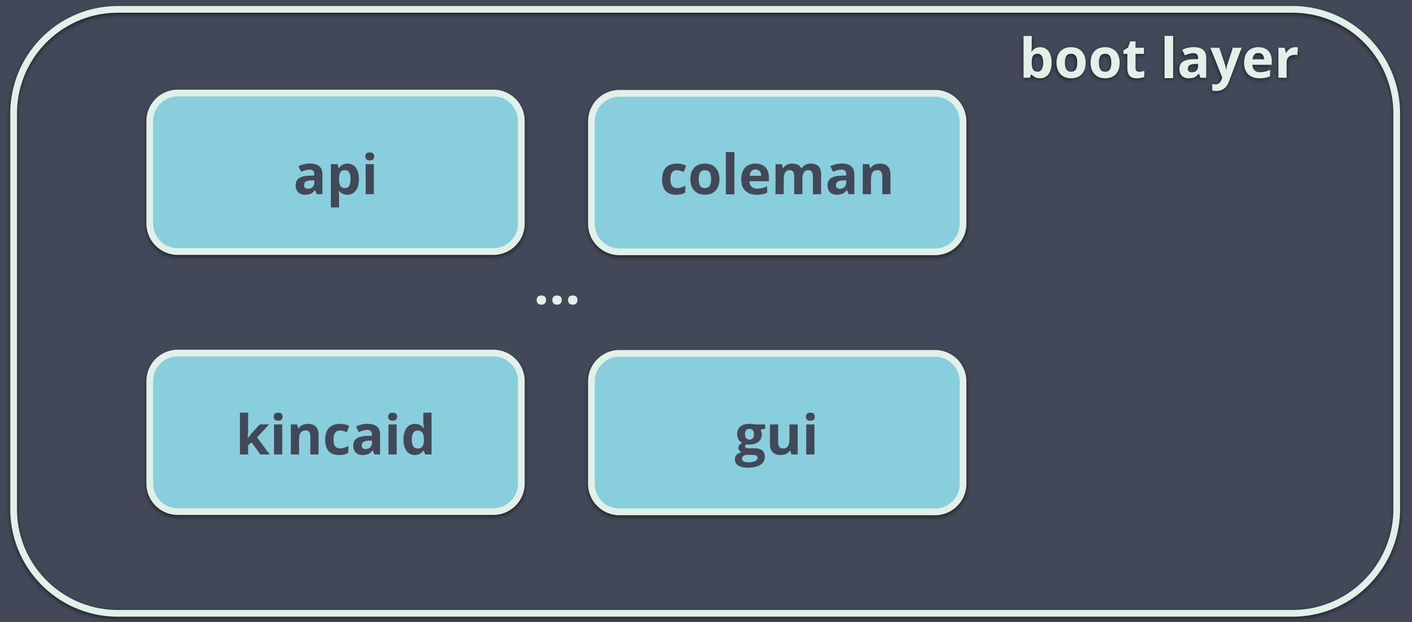
ModuleLayer

Immutable

App & platform modules

One version of a module

No dynamic loading?



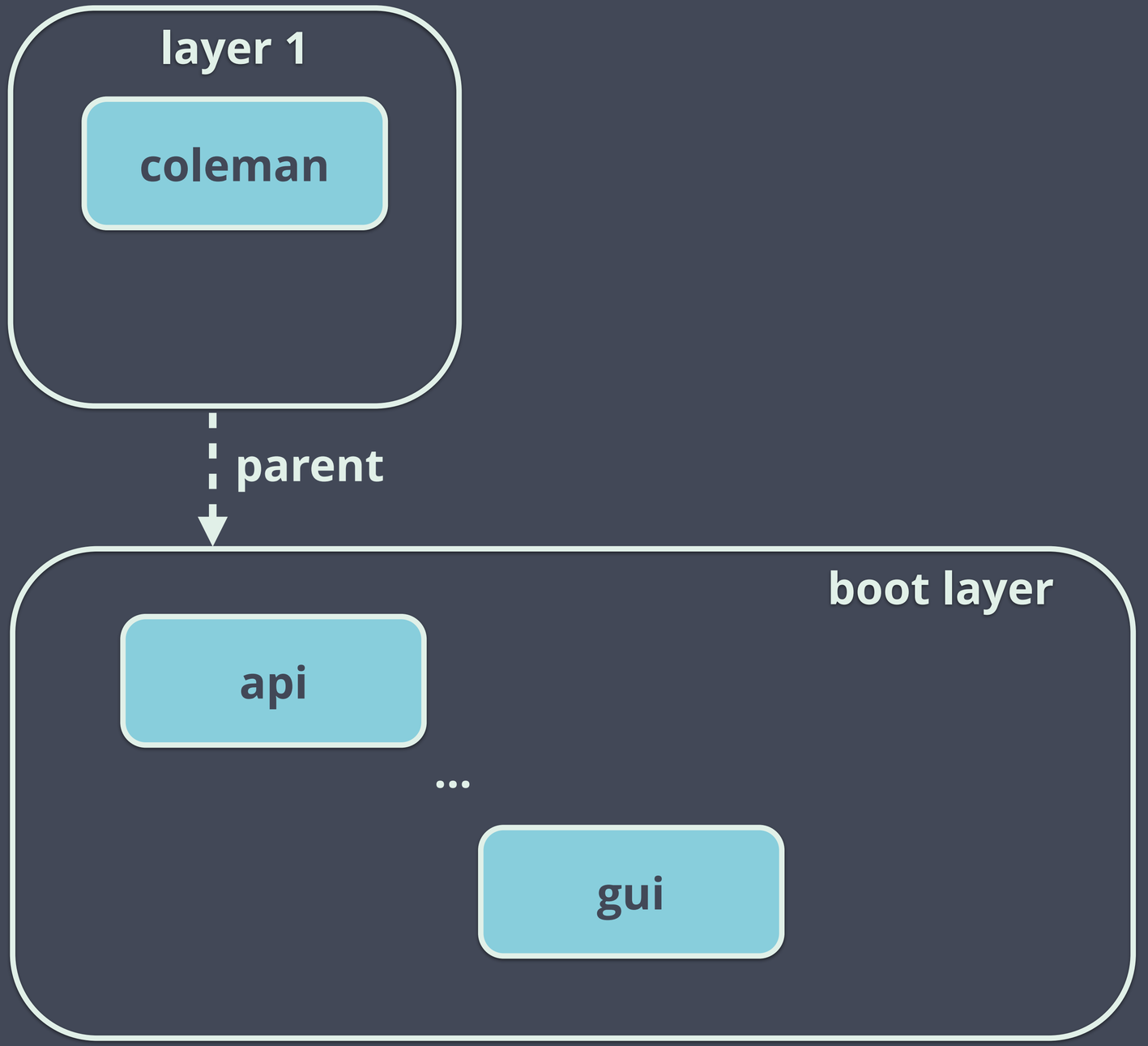
ModuleLayer

Create layers at run-time

Layers form hierarchy

Multiple versions in layers

ModuleLayer



Create layers at run-time

Layers form hierarchy

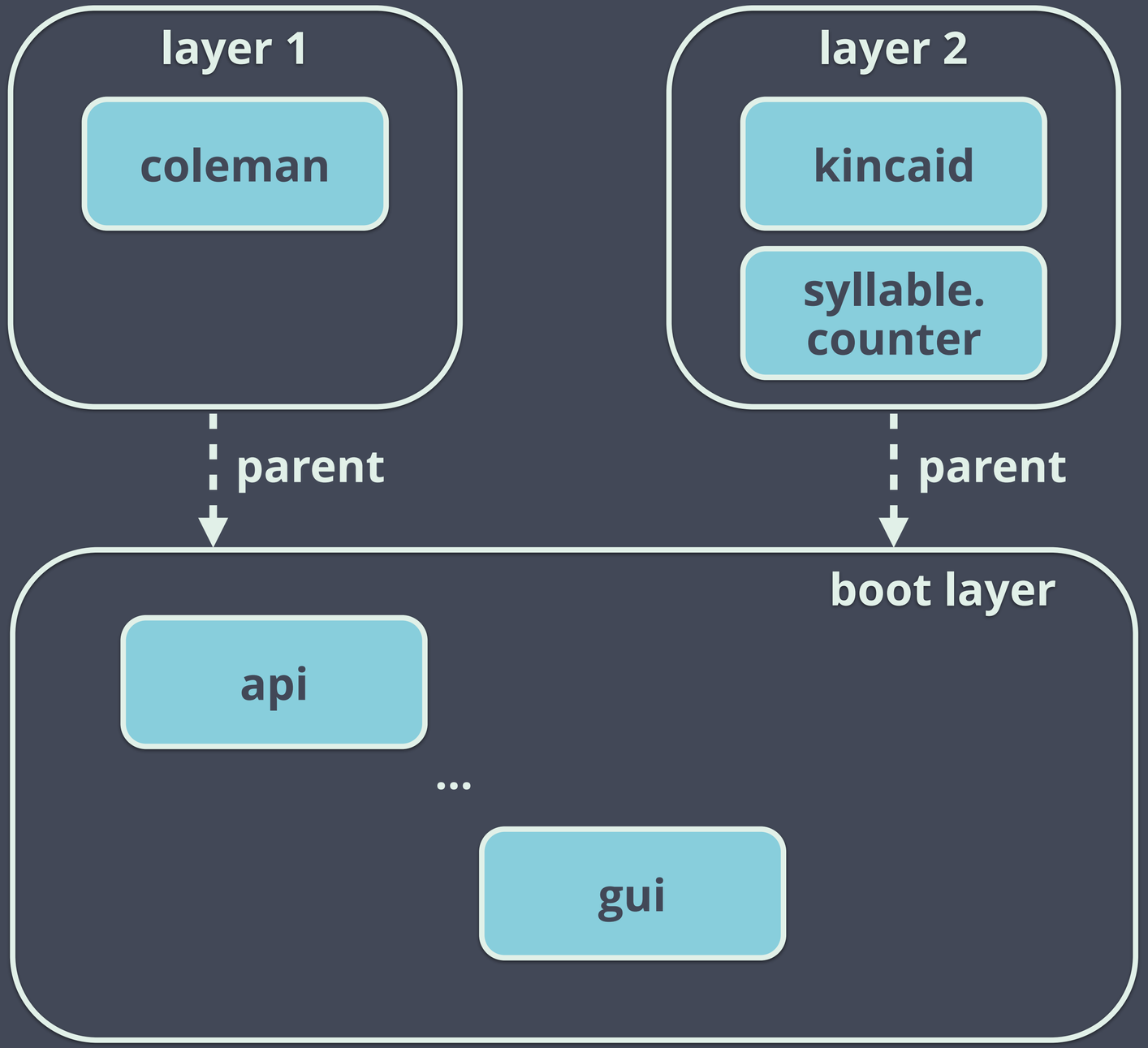
Multiple versions in layers

ModuleLayer

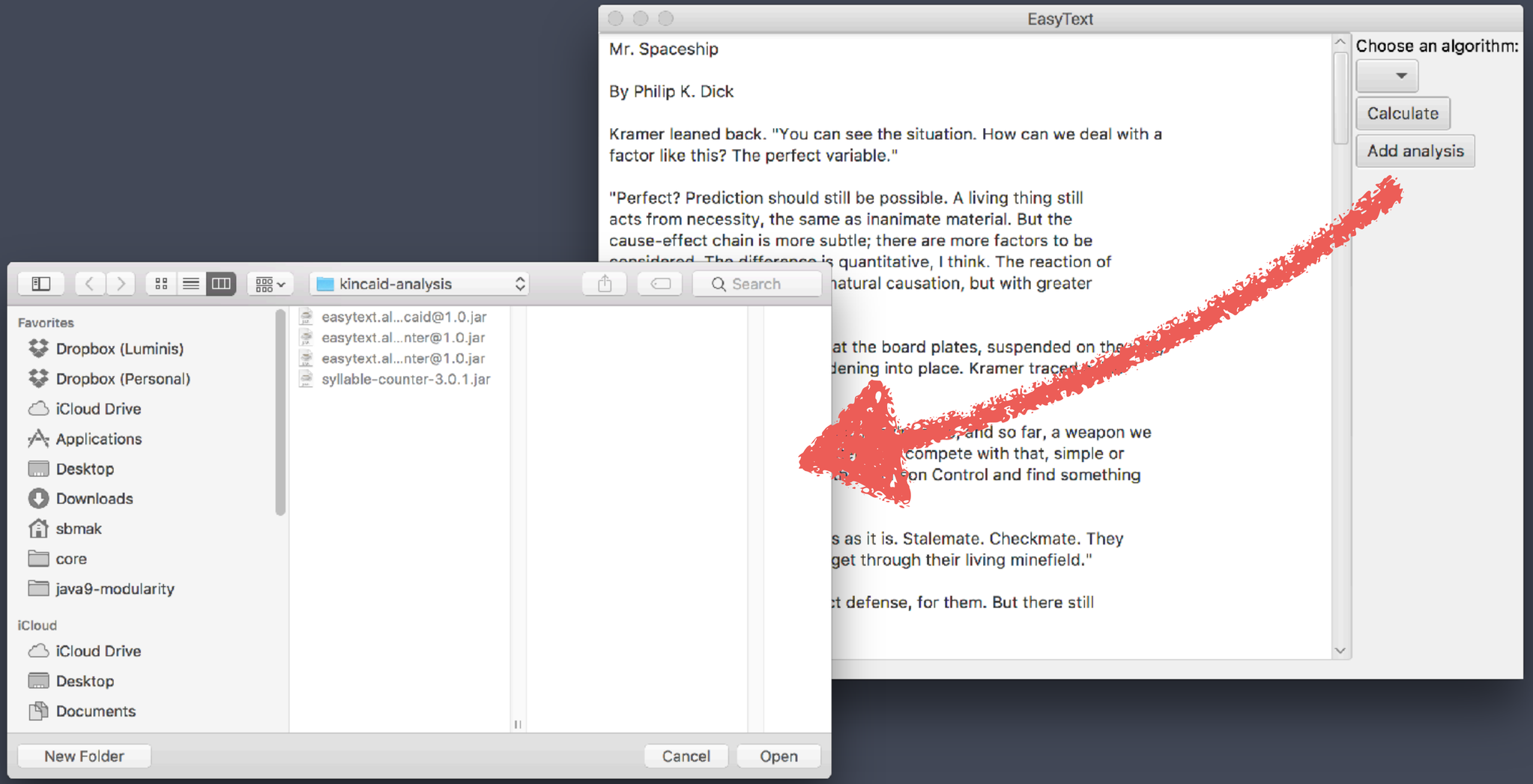
Create layers at run-time

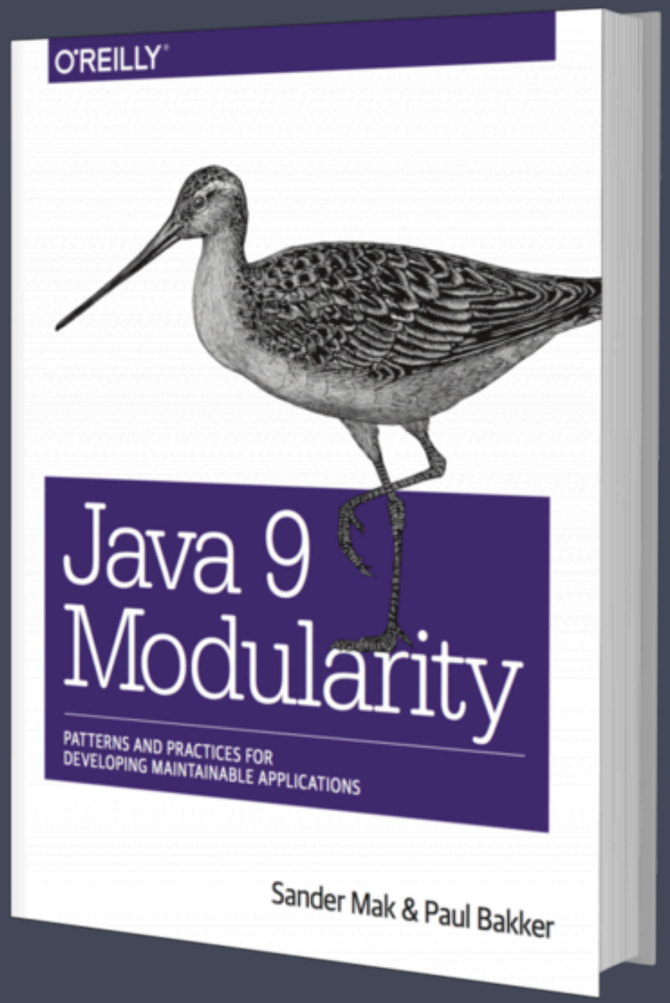
Layers form hierarchy

Multiple versions in layers



ModuleLayer Demo





Thank you.

luminis
Conversing worlds

@Sander_Mak

PLURALSIGHT

javamodularity.com

bit.ly/sander-ps