Software Design & Programming Techniques

Frameworks and Libraries

Prof. Dr-Ing. Klaus Ostermann

Based on slides by Prof. Dr. Mira Mezini

Frameworks and Libraries

- 6.1 Frameworks vs. Design Patterns vs. Applications vs. Libraries
- 6.2 Library Design Principles
- 6.3 Customizing Frameworks
- 6.4 Inversion of Control
- 6.5 Dependency Injection
- 6.6 Case Study: Log4J
- 6.7 Strengths and Weaknesses of Frameworks

6.1 Frameworks vs. Design Patterns vs. Applications vs. Libraries

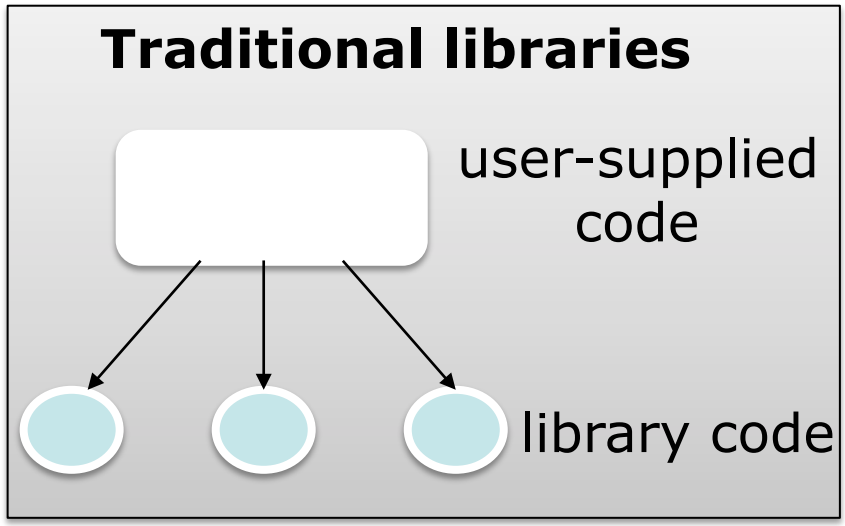
What is an (OO) Framework?

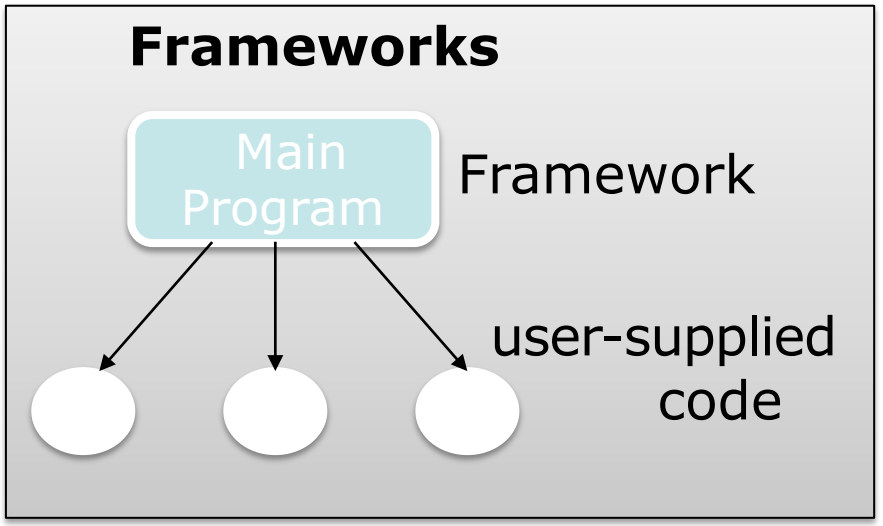
- A ***set of cooperating classes*** that makes up a reusable design ***for a specific class of software***.
- A framework provides architectural guidance by ***partitioning the design into abstract classes*** and defining their ***responsibilities and collaborations***.
- A developer customizes the framework to a particular application by ***subclassing*** and ***composing*** instances of framework classes. That's why frameworks are often called ***semi-complete applications*.**
- A framework solves problem in a ***particular problem domain***.
  See next slide for examples.

What is a library?

- A set of reusable coherent programming abstractions (classes, methods, functions, data structures)
- Focus on black-box reuse
- A library can also be seen (and used as) a domain-specific language

Libraries vs. Frameworks





- ***Control flow*** is ***dictated by*** the ***framework*** and is the same for all applications.

- The framework is the main program in coordinating and sequencing application activity. i.e., it manages the object lifecycle

Libraries vs Frameworks

- „Traditional“ difference: Who is in charge of the control flow
- However, this difference is only well-defined if one considers libraries that can only be parameterized by first-order values
- Libraries that accept higher-order parameters (such as first-class functions or objects) are quite similar to frameworks
  - Similar inversion of control
- Remaining difference: Frameworks are often white-box or grey-box whereas libraries are more black-box
  - Frameworks can be adapted in more ways, also ways not anticipated by the framework developer
  - Library developers must anticipate every extension point, but in turn libraries can be changed more easily without invaliding clients
- No strict discrimination between the two terms possible

Frameworks vs. Design Patterns

- So what is the difference between both frameworks and design patterns?

Recap, a Pattern is…

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

- Christopher Alexander

A Design Pattern is … (continued)

Design Pattern. A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems.

It describes the problem, the solution, when to apply the solution, and its consequences.

The solution is a general arrangement of objects and classes that solve the problem.

The solution is customized and implemented to solve the problem in a particular context.

- GoF

Frameworks vs. Design Patterns

- Sounds similar (at least partially), right?
- So again, so what is the difference between a framework and a design pattern?

Frameworks vs. Design Patterns

- **Patterns are smaller** than frameworks.
  - A framework contains many patterns (Visitor, Decorator etc.).
  - The opposite is not true.
- **Patterns are language independent.**
  - Patterns solve OO language issues (Java, C++, Smalltalk).
  - Frameworks are written in a specific programming language.
- **Patterns are more abstract** than frameworks.
  - Patterns do not solve application domain specific problems.
  - Frameworks provide support for a particular application domain. Frameworks provide reusable code

Frameworks vs. Design Patterns

Frameworks describe:

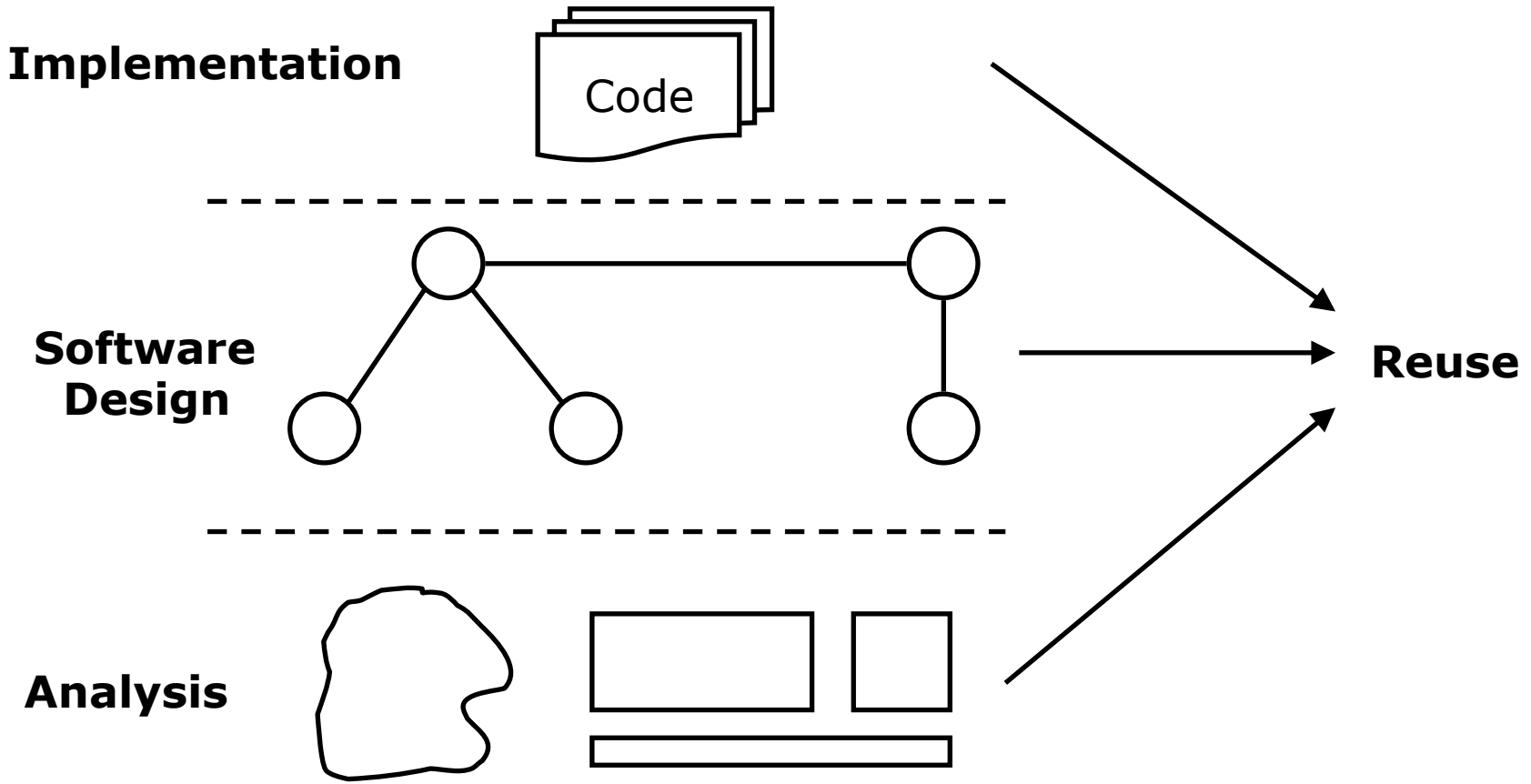
- the ***interface of each object*** and the ***flow of control between*** them.
- how the ***responsibilities*** are mapped onto its objects

In other words:

- A Framework provides ***architectural guidance***
- by ***partitioning*** the ***design into abstract classes*** and
- ***defining*** their ***responsibilities and collaborations***.

The high level design is the main intellectual content of software, and frameworks are a way to reuse it!

Levels of Reuse with Frameworks



A Framework is not...

- **... a design pattern.**
  - patterns describe ideas and perspectives;
  - frameworks are implemented software.

- **... an application.**
  - frameworks do not necessarily provide a default behavior, hence they are not executable programs;
  - They can be perceived as a partial design but they do not describe every aspect of an application.

- **... a class library.**
  - applications that use classes from a library invoke predefined methods, whereas frameworks invoke predefined methods supplied by the user.
    → see section about inversion of control for details…
  - But see earlier discussion about libraries vs. frameworks

6.2 Library Design Principles

Libraries

- The oldest, most common, and most successful way of reusing code
- Languages are designed to support libraries
  - Works together with static typing, import/export mechanisms, separate compilation, …
- If you have the choice of achieving your reuse goal with libraries or with some other mechanism, then libraries are typically the best choice
  - Composability with other libraries
  - Support by type and module system
  - Information hiding, substitutability, …
- But libraries need a good design to be useful!

Basic Library Design Principles

- Libraries should be as context-independent as possible
  - Every context dependency limits reusability
- Context dependencies (e.g. on other libraries) should be expressed via interfaces
  - Leaves more freedom to library users
- Libraries should have a clean, well-defined scope
- Library should have a well-defined interface
  - To make black-box usage possible
  - Interface should be cleanly separated from implementation details
    - E.g. via separate packages
- Library designer has to think about variability points of the library
- Different form of variability
  - Parameterization by values
  - Parameterization by types
  - Parameterization by functions/closures or objects

Issues in Library Design

- Simulating a domain-specific syntax
  - Depends on syntactic flexibility of host language
    - E.g., possibility to use operators, prefix/infix/postfix notation etc.
- Domain-specific optimizations
  - Can be difficult to achieve with traditional libraries
  - Idea of „active library“

6.3 Customizing Frameworks

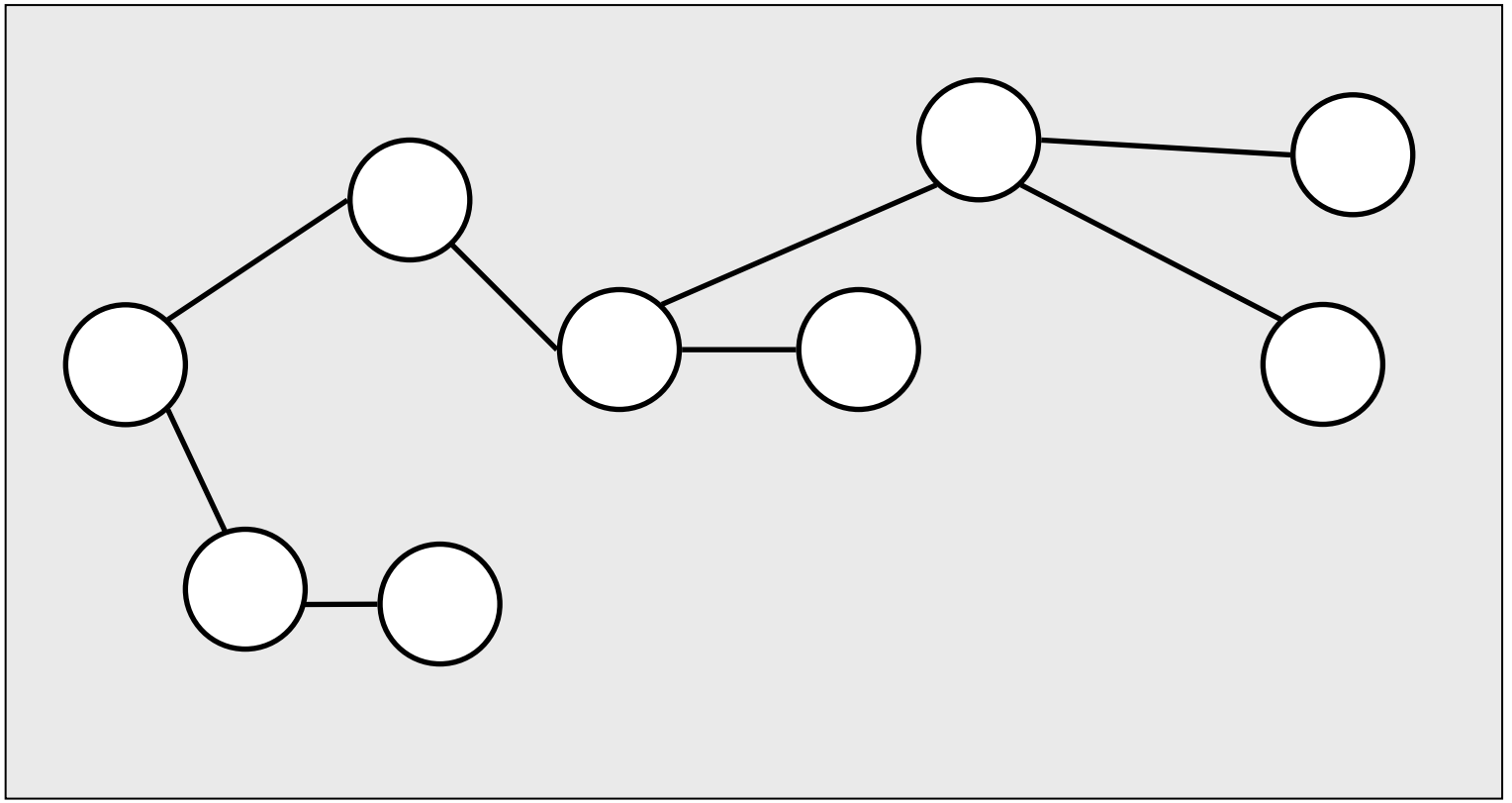
Customization Points

- So far, we talked about frameworks being semi-complete applications that developers need to extend to make them work as application. Thus, the question arises ***how*** one can customize a framework.

- So far we have learned that frameworks have an architecture and a design that is reused by application developers. Let's consider following collection of nodes and links to represent a framework …

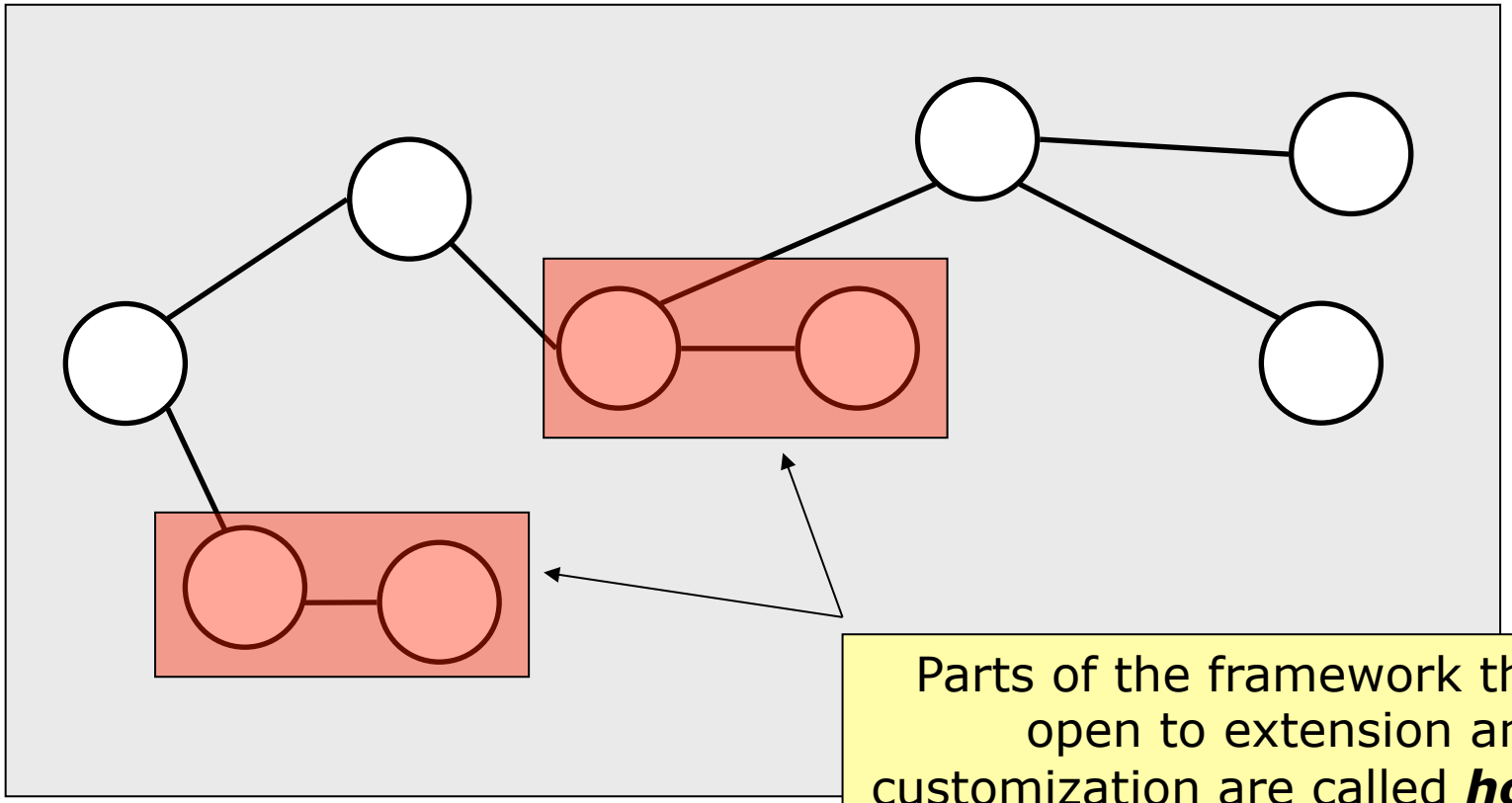
Simplified Representation of a Framework

- Nodes represent classes, links between nodes represent associations between classes used for collaboration between classes.



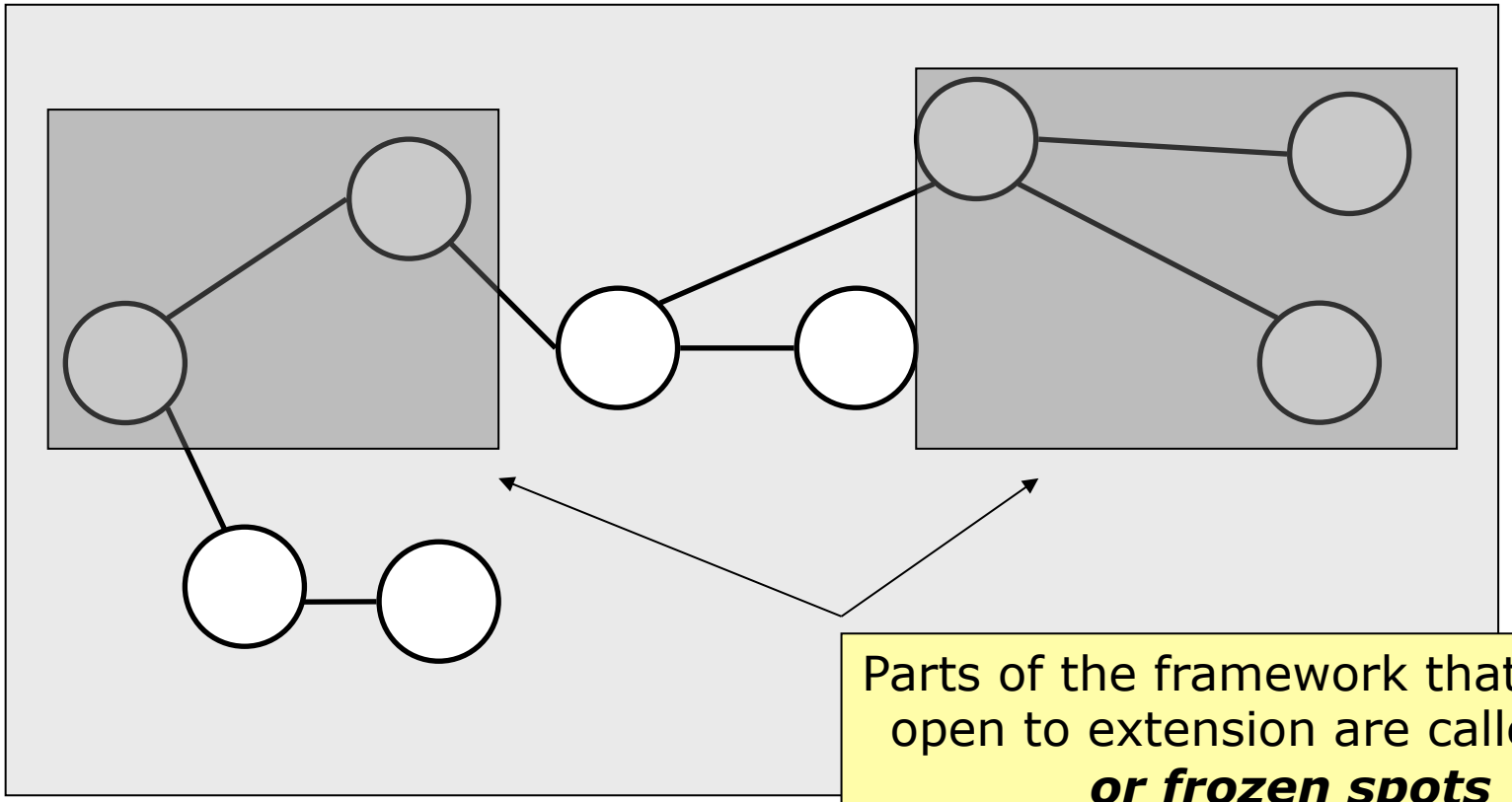
Framework Hot Spots

- Since frameworks are incomplete there must be some points in the design allowing a developer to extend the framework. This extension points are called ***hot spots***.



Framework Frozen Spots

- Not all parts of the framework are necessarily designed for being extensible. These non-extensible spots are called ***frozen spots***.



Parts of the framework that are **not** open to extension are called ***cold or frozen spots***

How to extend a framework concretely?

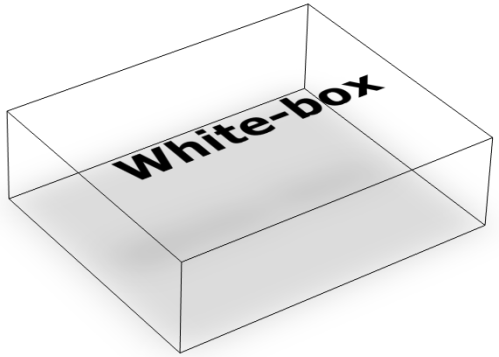
- You learned that there are some parts that can be extended and some can’t. But how do you do that actually?

- The short answer: It depends. Before explaining that, we need to introduce another classification for frameworks (additionally to the classification by their application domain).

Framework Classification By Extension Technique

Frameworks can (also) be classified by the **techniques used to extend them**. We distinguish between three different kinds of frameworks:

White-box Frameworks



- White-box frameworks are customized by **subclassing** existing framework classes.

- Subclassing requires detailed knowledge:
  - Component interfaces of the class.
  - Flow of control in the new component .
  - Overriding predefined hook methods → later…

- **Learning white-box frameworks is hardest but most powerful way.**

Extension Example

```
public class MyWizard extends Wizard {

    @Override
    public void addPages() {
        // TODO Auto-generated method stub
        super.addPages();
    }

    @Override
    public boolean performFinish() {
        // TODO Auto-generated method stub
        return false;
    }
}
```

- One way is extending a framework base class - maybe this extension uses the template method pattern.

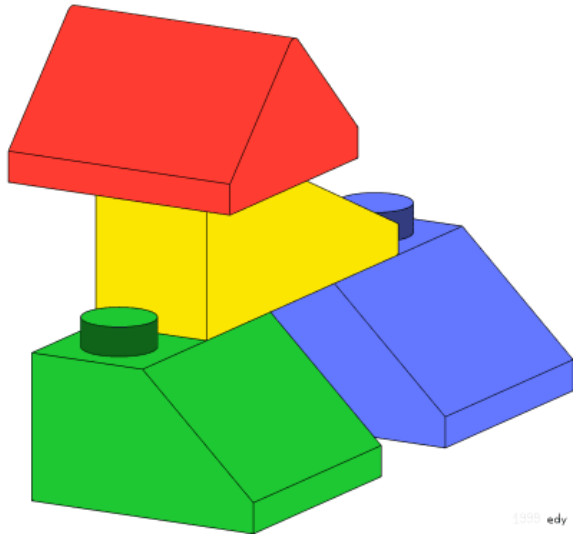
Black-box Frameworks

- Black-box frameworks are customized using already **existing components.**
- Black-box requires less programming:
  - Connecting existing components only.
  - Writing of new classes is not required.
- Black-box frameworks are less flexible.
  - Usability depends on component library.
- **Black-box frameworks are easier to learn.**

Extension Example

- In black-box frameworks you may observe the same patterns as in white-box frameworks. But the main difference is: you don't provide the implementations for these components – you just reuse them and plug them together as you need it.
- Technical difference: Object composition (black-box reuse) vs. subclassing (white/grey/black-box reuse, depending on the subclass interface description)

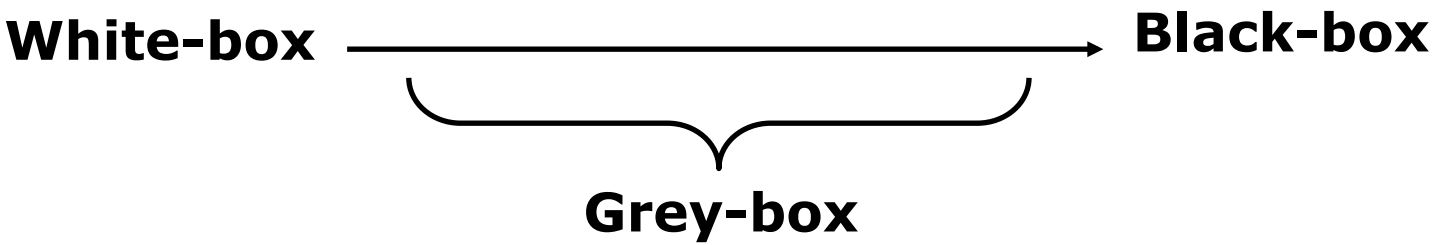
Like building a toy house from Legos …



Grey-box Frameworks

- Grey-box frameworks using both **parameterization** and **refinement**

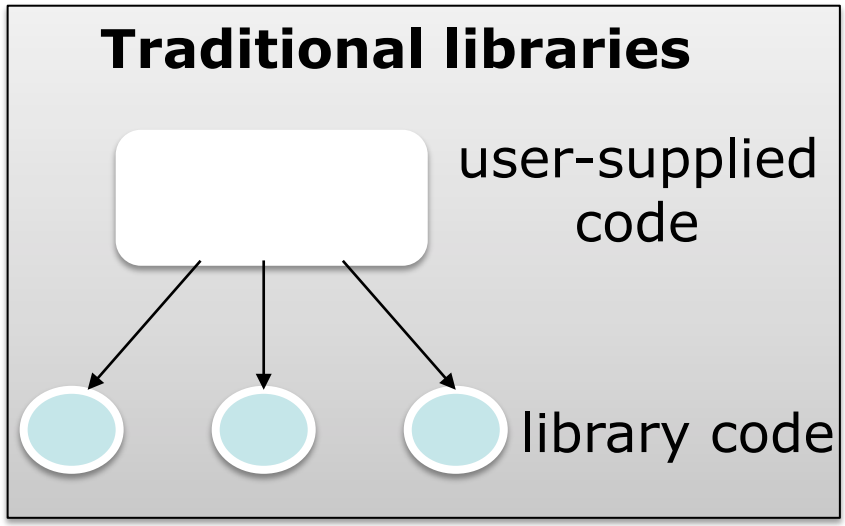
- Frameworks typically evolve from white-box to black-box frameworks over a number of iterations:

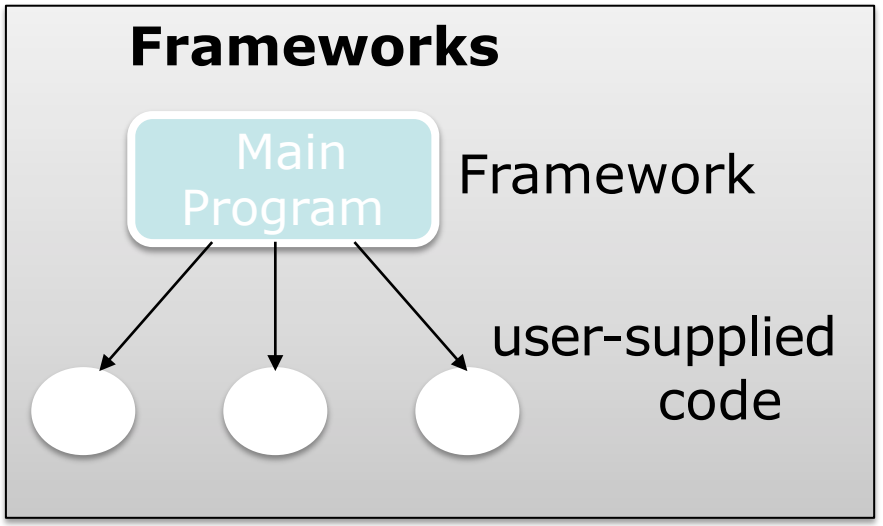


- However, it will be hard to find pure black-box frameworks. Typically, they contain a few white-box elements too.

6.4 Inversion of Control or …

Libraries vs. Frameworks



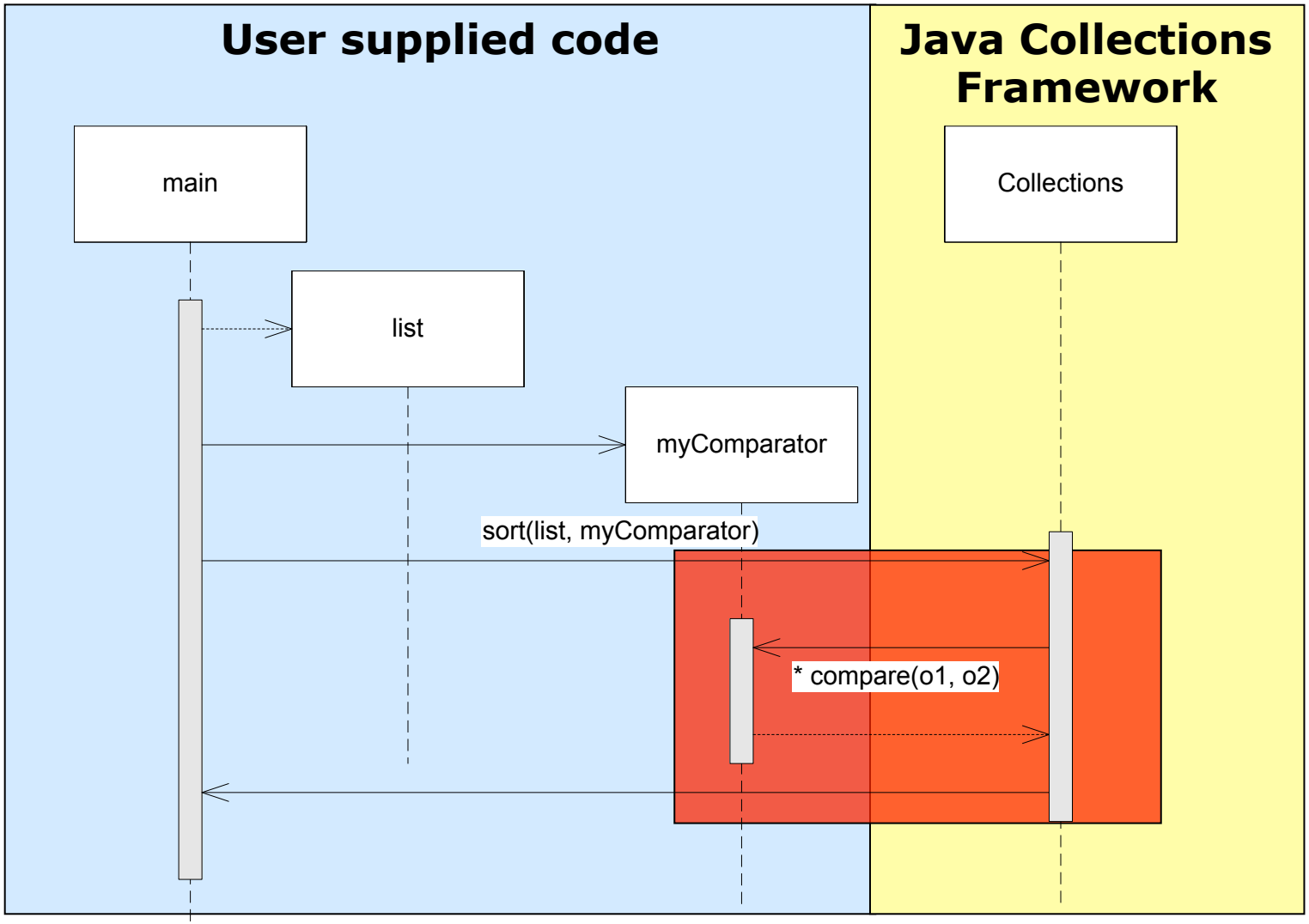


- ***Control flow*** is ***dictated by*** the ***framework*** and is the same for all applications.

- The framework is the main program in coordinating and sequencing application activity. i.e., it manages the object lifecycle

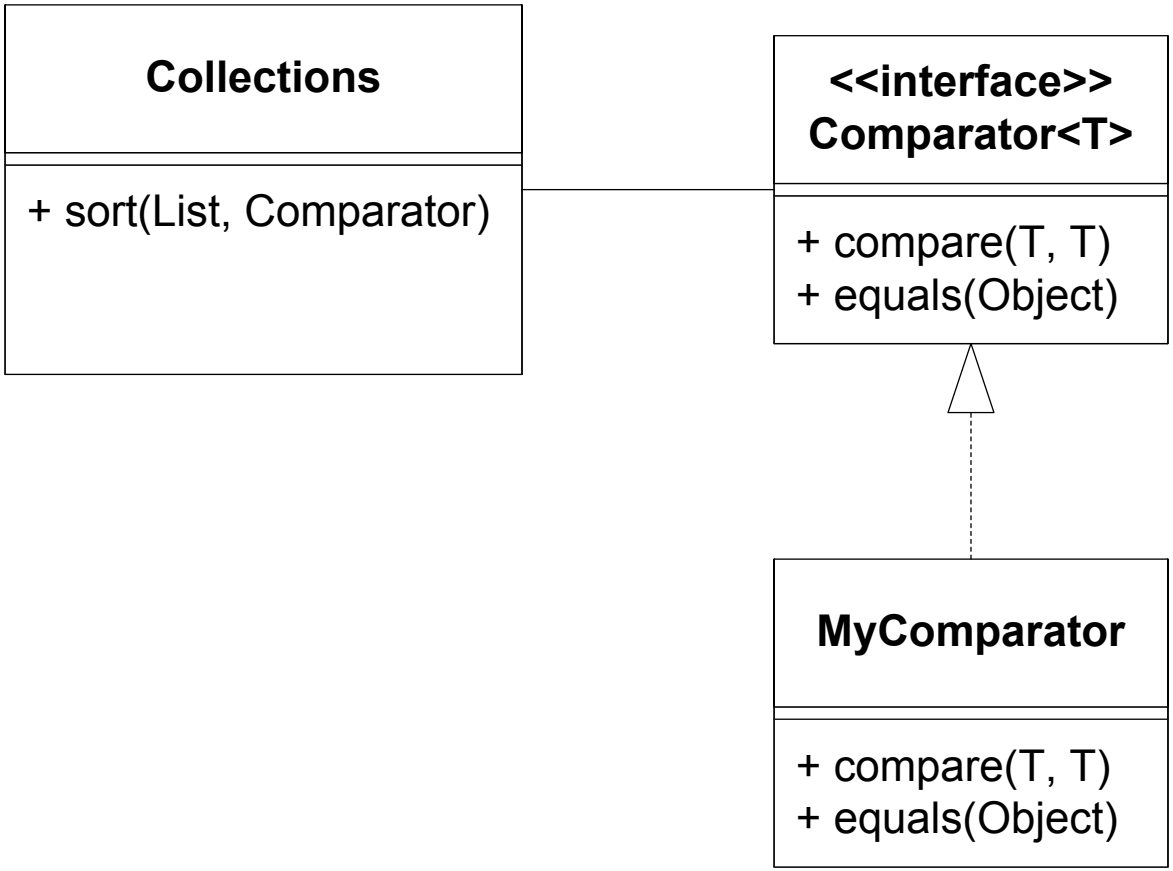
Small Example of IoC in Action

```
Collections.sort(list, new MyComparator());
```



Dependency Inversion in Frameworks

- Dependency Inversion is the most essential principle applied on frameworks .



Dependency Inversion in Functional Languages

- Sorting in Haskell:

Dependency Inversion by Higher-Order Function:

```
sort :: (a -> a -> Bool) -> [a] -> [a]
```

Example: `sort (\x y -> x > y) [3,6,2]`

Dependency Inversion with Type Classes

```
sort :: Ord a => [a] -> [a]
```

Example:

```
instance Ord Int where
  a <= b = a > b

sort [3,6,2]
```

6.5 Dependency Injection

Motivation

Given:

- We have many components and want to build an application out of them.
- We can decrease coupling by good OO practices such as programming against interfaces, registries, etc.
- However, most components collaborate with other components or need to have access to resources.

Questions:

- How can we **minimize the coupling** between components**,** between a component and the environment, between a component and its required services?
- How can we improve the reuse potential?
- How can we achieve a better testability of our components?

Developing A tweets client

From a "normal" design to Dependency Injection (DI)

Steps:

- Setting the stage
- Constructors
- Factories
- Dependency Injection
  - by hand
  - with Google Guice

Code you might write

A tweets client

```
public void postButtonClicked() {
   String text = textField.getText();

   if (text.length() > 140) {
     final Shortener shortener = new TinyUrlShortener();
     text = shortener.shorten(text);
   }

   if (text.length() <= 140) {
     final Tweeter tweeter = new SmsTweeter();
     tweeter.send(text);
     textField.setText("");
   }
 }
```

Problems with this solution?

- The **`TweetClient`** depends on two components:
  - a **`Shortener`** (namely, a **`TinyUrlShortener`**) for shortening text messages that are too long, and
  - a **`Transport`** (namely, a **`SmsTweeter`**) that sends the message to, say, a Twitter server.

- How about testability?
  - You may have noticed that the code actually builds its dependencies immediately, i.e, we call constructors of **`TinyUrlShortener`** and **`SmsTweeter`** directly in our code.
  - This is really convenient and it is really terse but there's a lot of problems with it. Most notably, this code doesn't lend itself to testing because of the hardcoded dependencies!

Getting dependencies via their constructors

...calling new directly doesn't afford testing

```
public void postButtonClicked() {
    String text = textField.getText();

    if (text.length() > 140) {
      final Shortener shortener = new TinyUrlShortener();
      text = shortener.shorten(text);
    }

    if (text.length() <= 140) {
      final Tweeter tweeter = new SmsTweeter();
      tweeter.send(text);
      textField.setText("");
    }
  }
```

We post to tinyurl.com and send an SMS for each test! This is neither fast nor reliable.

Getting Dependencies from factories

```
public void postButtonClicked() {
   String text = textField.getText();

   if (text.length() > 140) {
     final Shortener shortener = ShortenerFactory.get();
     text = shortener.shorten(text);
   }

   if (text.length() <= 140) {
     final Tweeter tweeter = TweeterFactory.get();
     tweeter.send(text);
     textField.setText("");
   }
 }
```

Factories come to rescue. But they introduce another problem …

Implementing the factory

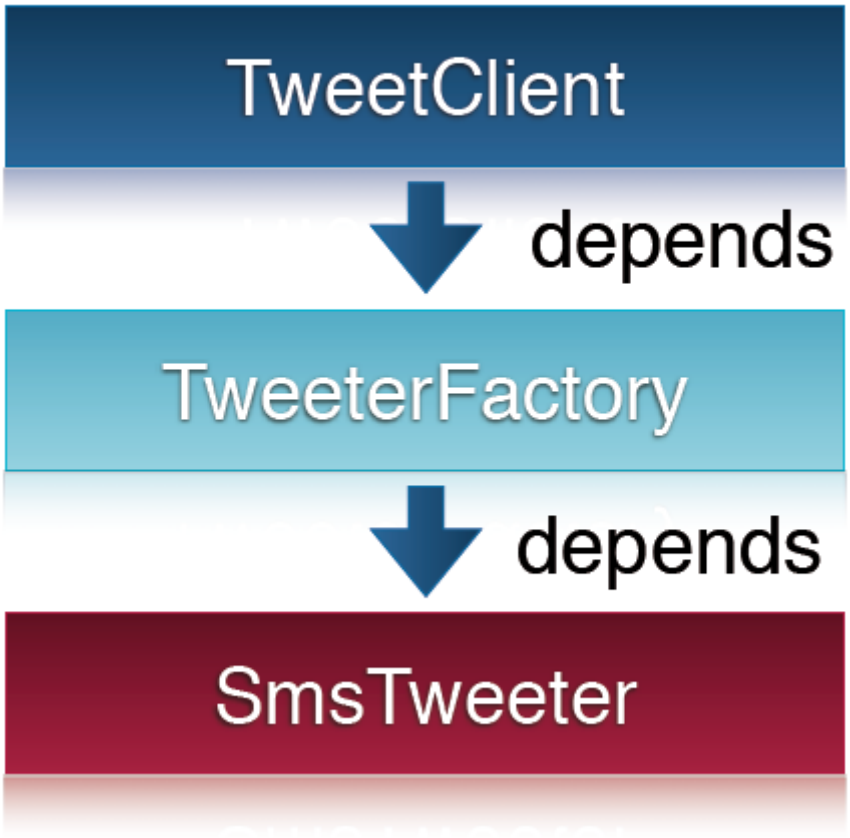
All of this boilerplate slows you down.

```
public class TweeterFactory {
  private static Tweeter tweeter;

  public static Tweeter get() {
    if (tweeter == null) {
      tweeter = new SmsTweeter();
    }
    return tweeter;
  }

  public static void setForTesting(Tweeter testTweeter) {
    tweeter = testTweeter;
  }
}
```

We still have to write a factory for the URL shortener…

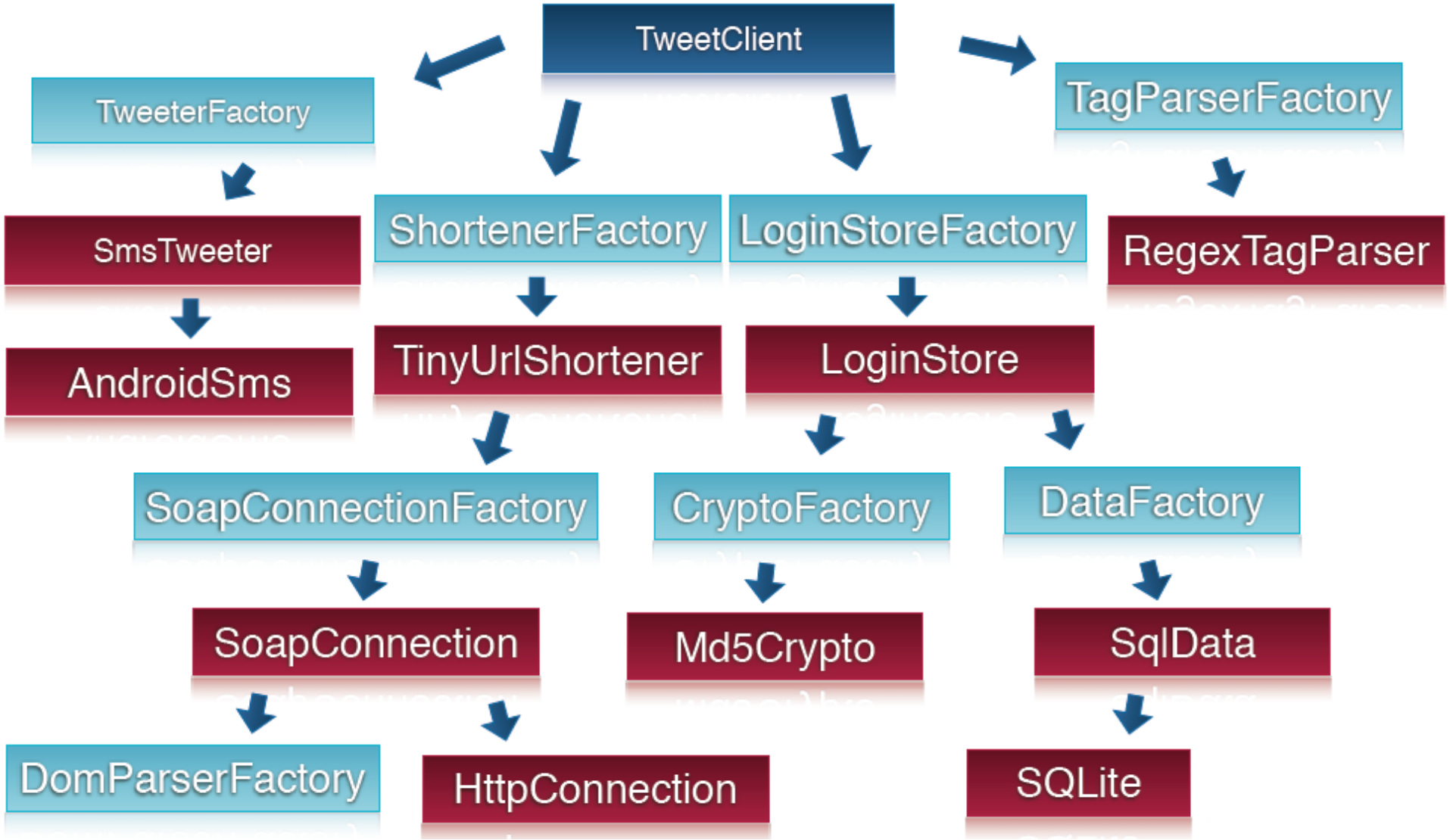
Factory dependence graph

Design causes a deep net of dependencies...



Factory dependency graph

...design applied recursively



Testing your code with factories

Using shared mutable factories is error prone…

```
@Test
public void testTweet() {

  // setup
  final String message = "Hello!";
  final TweetClient tweetClient = new TweetClient();
  final MockTweeter tweeter = new MockTweeter();
  TweeterFactory.setForTesting(tweeter);
  ...
  // exercise
  tweetClient.getEditor().setText(message);
  tweetClient.postButtonClicked();

  // verify
  assertEquals(message, tweeter.getSent());
}
```

Testing your code with factories

```
@Test
public void testTweet() {

  // setup
  final String message = "Hello!";
  final TweetClient tweetClient = new TweetClient();
  final MockTweeter tweeter = new MockTweeter();
  TweeterFactory.setForTesting(tweeter);
  ...
  // exercise
  tweetClient.getEditor().setText(message);
  tweetClient.postButtonClicked();

  // verify
  assertEquals(message, tweeter.getSent());

  // teardown
  TweeterFactory.setForTesting(null);
}
```

Don’t forget to clear the playground after your tests…

6.5.1 Dependency injection by hand

objects come to you

```
public class TweetClient{

  Shortener shortener;
  Tweeter tweeter;

  public TweetClient(Shortener shortener, Tweeter tweeter) {
    this.shortener = shortener;
    this.tweeter = tweeter;
  }

  public void postButtonClicked() {
    ...
    if (text.length() <= 140) {
      tweeter.send(text);
      textField.setText("");
    }
  }
}
```

Dependency Injection: rather than looking it up, get it passed in.

Testing with dependency injection

no cleanup required…

```
public void testSendTweet() {
    MockShortener shortener = new MockShortener();
    MockTweeter tweeter = new MockTweeter();
    TweetClient tweetClient
                = new TweetClient(shortener, tweeter);
    tweetClient.getEditor().setText("Hello!");
    tweetClient.postButtonClicked();
    assertEquals("Hello!", tweeter.getSent());
}
```

However, we still have to provide create the TweetClient, right?

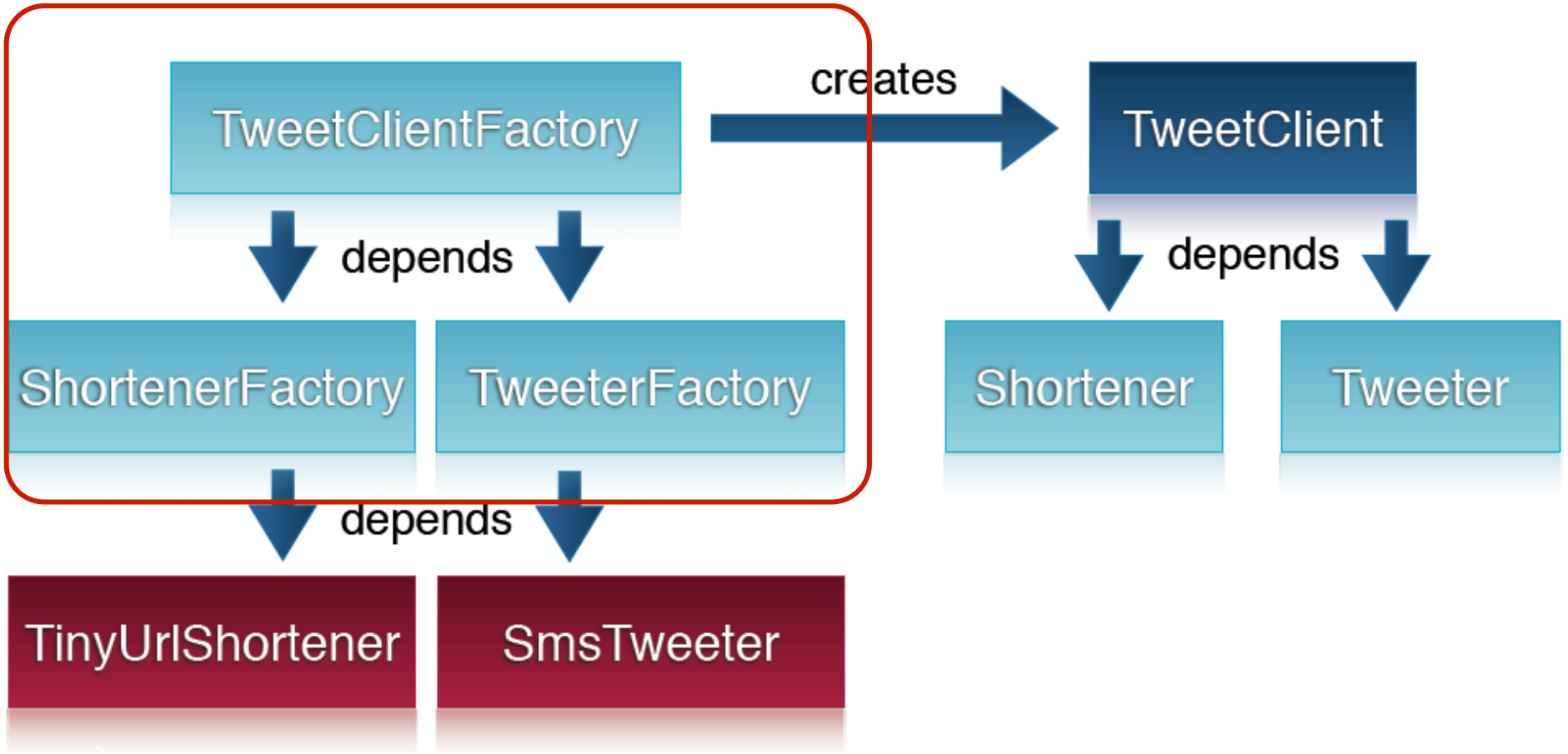
Where does the dependency go?

```
public class TweetClientFactory {

    private static TweetClient testValue;

    public static TweetClient get() {

        if (testValue != null) {
            return testValue;
        }

        Shortener shortener = ShortenerFactory.get();
        Tweeter tweeter = TweeterFactory.get();
        return new TweetClient(shortener, tweeter);
    }
}
```

DI motto: Push dependencies from the core to the edges

Where does the dependency go?

your application code sheds its heavyweight dependencies

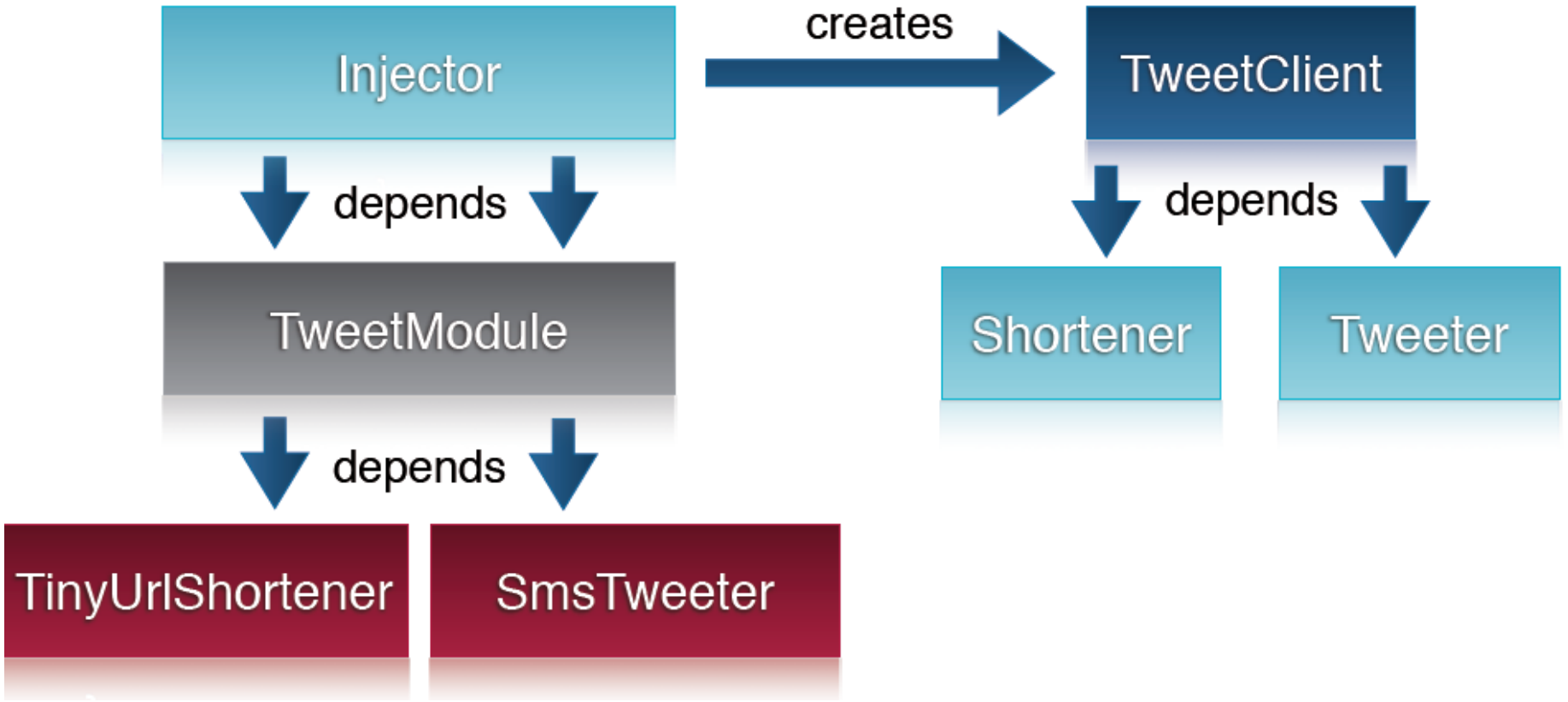


Recap

So what are your goals?

- Keep as flexible as possible which components to use at runtime, i.e., reduce any hard-coded dependencies in production code.
- Separate the glue code from the component code
- Can be done by hand, or with the help of DI inversion tools such as Guice

Dependency Injection with Guice



Configuring the injector using modules

```
import com.google.inject.AbstractModule;

public class TweetModule extends AbstractModule {

    protected void configure() {
        bind(Tweeter.class).to(SmsTweeter.class);
        bind(Shortener.class).to(TinyUrlShortener.class);
    }
}
```

Telling Guice to use your constructor

annotate a constructor with @Inject

```
import com.google.inject.Inject;

public class TweetClient {

    private final Shortener shortener;
    private final Tweeter tweeter;

    @Inject
    public TweetClient(Shortener shortener, Tweeter tweeter) {
        this.shortener = shortener;
        this.tweeter = tweeter;
    }
…
```

Bootstrapping Guice

```
public static void main(String[] args) {

    Injector injector =
            Guice.createInjector(new TweetModule());

    TweetClient tweetClient =
            injector.getInstance(TweetClient.class);


    tweetClient.show();
}
```

the DI framework creates all dependencies for you.

Bootstrapping Guice for Testing

Create a test configuration:

```
import com.google.inject.AbstractModule;

public class TweetTestModule extends AbstractModule {

    protected void configure() {
        bind(Tweeter.class).to(MockTweeter.class);
        bind(Shortener.class).to(MockShortener.class);
    }
}
```

Bootstrapping Guice for Testing

And use it in your tests…

```
public void testTweet() {

    Injector injector =
            Guice.createInjector(new TweetTestModule());

    TweetClient tweetClient =
            injector.getInstance(TweetClient.class);


    tweetClient.getEditor().setText("Hello!");
    tweetClient.postButtonClicked();
    assertEquals("Hello!", tweeter.getSent());}
```

Guice Recap

- Helps in separating wiring from component code
- Code becomes short

- There are also disadvantages
  - Loss of static type safety
  - What if a more flexible mapping from interfaces to classes is needed?
    - E.g., not a global mapping but mapping on a per-case basis?
    - Guice offers no support for these cases
  - Reflection is slow – this may or may not be a problem

Inversion of Control vs. dependency injection?

- These two terms are not really opposed to one another as the heading suggests.
- You will come across the term IoC quite often, both in the context of dependency injection and outside it. The phrase IoC is rather vague and connotes a general reversal of responsibilities how to obtain dependent-on components.
- DI is one instance of IoC

Terms & Definitions

- ***Hollywood Principle:***
  - The idea that a dependent is contacted with its dependencies
- ***Dependency injector:***
  - A framework or library that embodies the Hollywood Principle
- ***Dependency injection:***
  - The range of concerns with designing applications built on these principles
- ***Inversion of Control Containers:***
  - DI frameworks are sometimes referred to as IoC containers

Kinds of Dependency Injections

- ***Constructor Injection:***

```
@Inject
public TweetClient(Shortener shortener, Tweeter tweeter) {
    this.shortener = shortener;
    this.tweeter = tweeter;
}
```

- ***Setter/Method Injection:***
  (if method is specified via some interface its called interface injection)

```
@Inject void setShortener(Shortener shortener) {
    this.shortener = shortener;
}
```

- ***Field Injection:***

```
@Inject Shortener shortener;
@Inject Tweeter tweeter;
```

6.7 Strengths and Weaknesses of Frameworks

Benefits of Using Frameworks

- ***Modularity***
  - volatile implementation details encapsulated behind stable interfaces
  - improves software quality by localizing the impact of design and implementation changes
  - localization reduces the effort required to understand and maintain existing software
- ***Reusability***
  - frameworks allow the reuse of domain knowledge, architecture and code
  - Reuse of components enhance quality, performance, reliability and interoperability

Benefits of Using Frameworks

- ***Extensibility***
  - Framework enhances extensibility by providing explicit hook methods.
  - Hook methods systematically decouple the stable interfaces and behaviors of an application domain from a particular context.
- ***Inversion of control***
  - IOC leads to reduced coupling between components
  - Increases testability

Weaknesses when using Frameworks

- **Learning curve**
  - it often takes several months become highly productive with a complex framework
- **Integratability**
  - Application development will be increasingly based on integration of multiple frameworks together with class libraries, legacy systems and existing components in one application
- **Maintainability**
  - As frameworks evolve, the applications that use them must evolve with them …
- **Efficiency**
  - In Terms of memory usage, system performance …