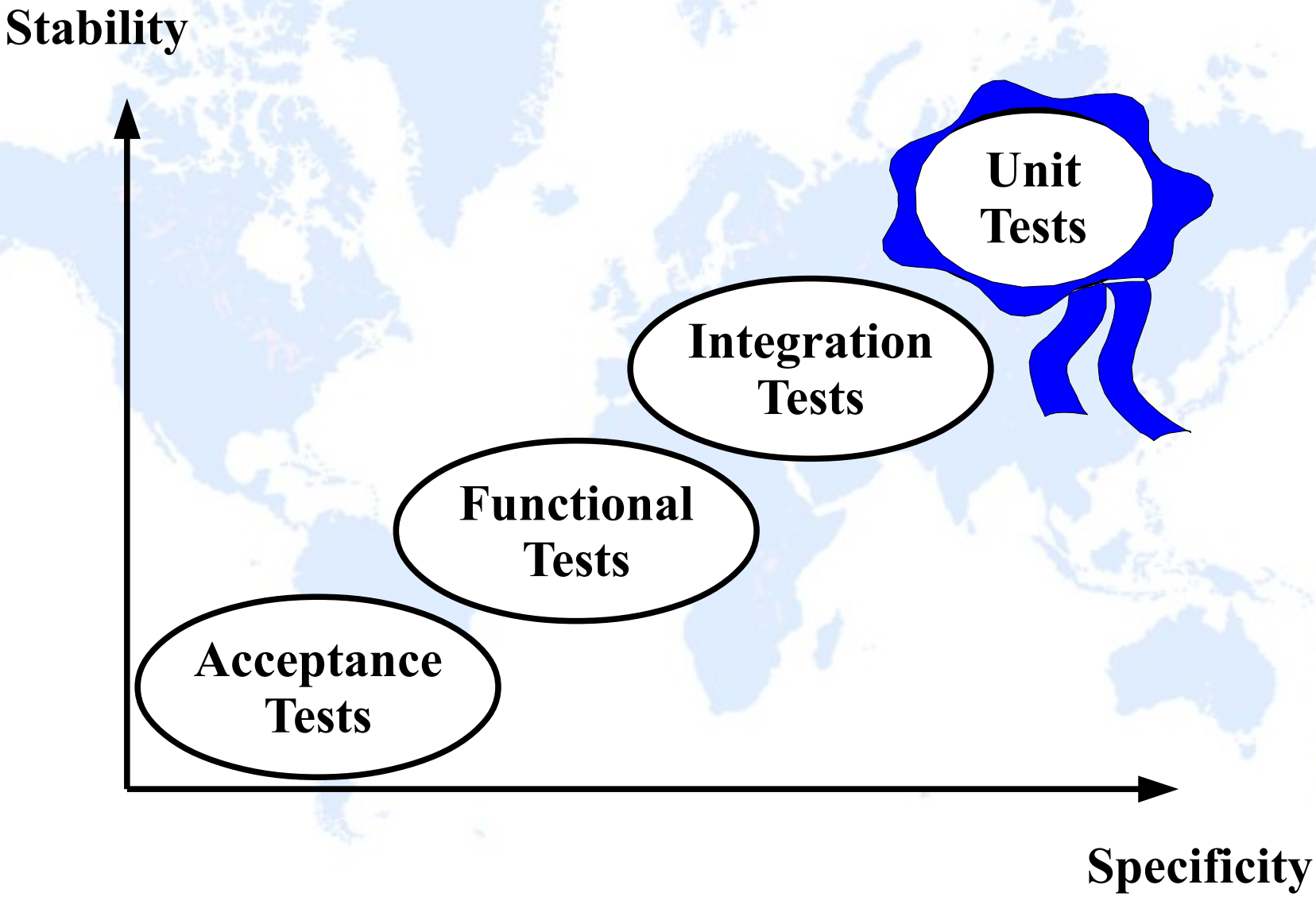
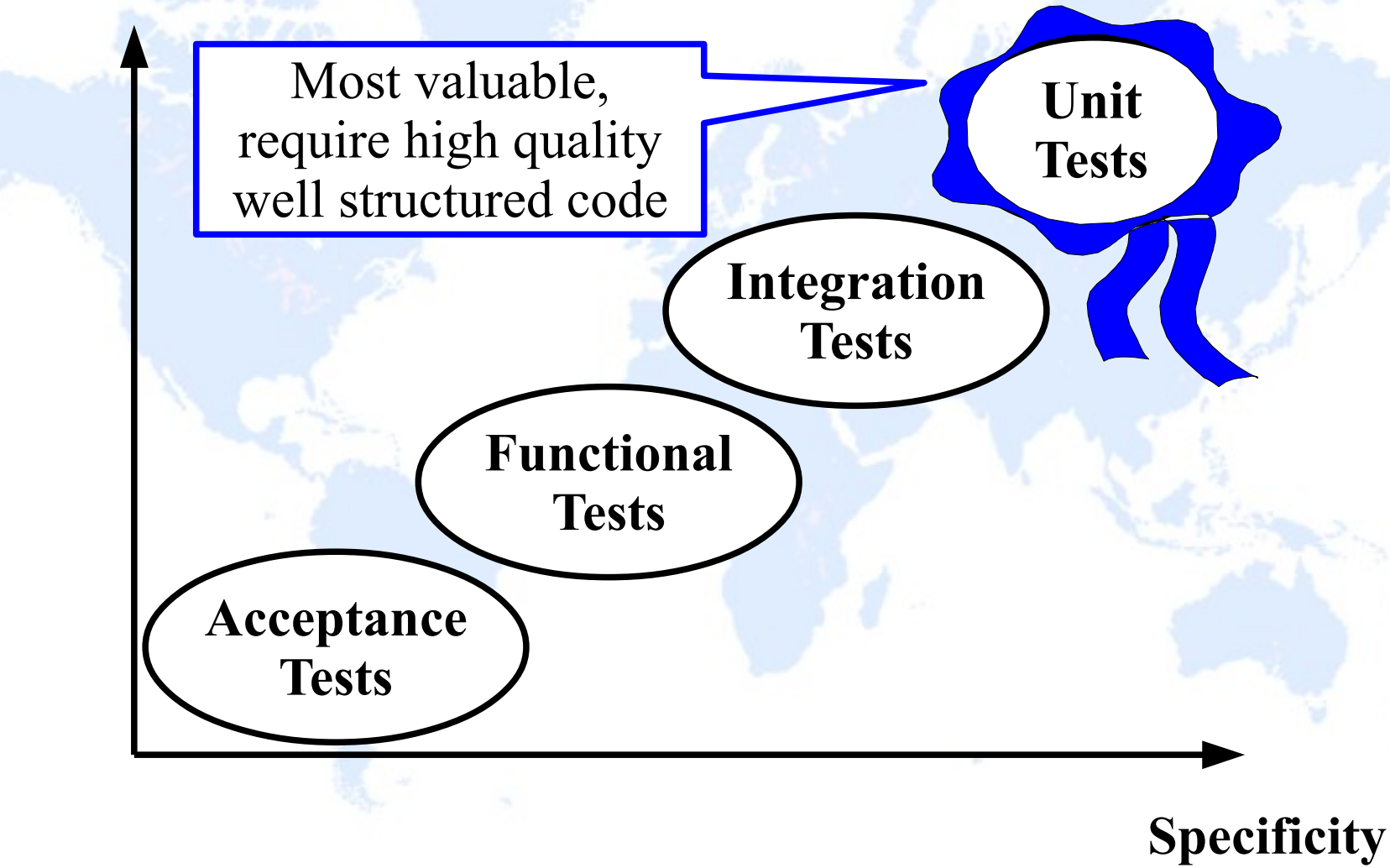
Dependency Injection

Not all tests are created equal



Not all tests are created equal



Real Unit Tests

- Test a single unit of code, i.e. a **single class**

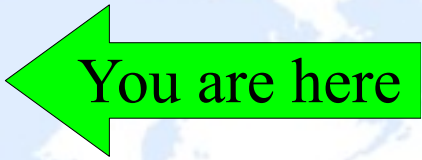
Real Unit Tests

- Test a single unit of code, i.e. a **single class**

We need to be able to
replace (mock/stub)
dependencies dynamically

Structure of this presentation

- Why do we want Dependency Injection?
- Code example: DI for generic PHP classes
- Code example: DI in Symfony 2



```php
<?php
use Guzzle\Http\Client;
use Acme\Logger\XmlLogger;

class FeedAggregator {
    private $client;
    private $logger;

    public function __construct () {
        $this->client = new Client();
        $this->logger = new XmlLogger();
    }

    public function retrieveFeed ($baseurl, $path) {
        $request = $this->client->setBaseUrl($baseurl)->get($path);
        $response = $request->send();
        if (200 != $response->getStatusCode()) {
            $this->logger->log('Could not get: '.$host.$path);
            return null;
        }

        return $response->getBody();
    }
    // ...
}
```

Why is this class difficult to unit test?

```php
<?php
use Guzzle\Http\Client;
use Acme\Logger\XmlLogger;

class FeedAggregator {
    private $client;
    private $logger;

    public function __construct () {
        $this->client = new Client();
        $this->logger = new XmlLogger();
    }

    public function retrieveFeed ($baseurl, $path) {
        $request = $this->client->setBaseUrl($baseurl)->get($path);
        $response = $request->send();
        if (200 != $response->getStatusCode()) {
            $this->logger->log('Could not get: '.$host.$path);
            return null;
        }

        return $response->getBody();
    }
    // ...
}
```

Why is this class difficult to unit test?

What if we want unit tests to run fast without waiting for the network?

```php
<?php
use Guzzle\Http\Client;
use Acme\Logger\XmlLogger;

class FeedAggregator {
    private $client;
    private $logger;

    public function __construct () {
        $this->client = new Client();
        $this->logger = new XmlLogger();
    }

    public function retrieveFeed ($host, $path) {
        $request = $this->client
        $response = $request->se
        if (200 != $response->ge
            $this->logger->log('Coul
            return null;
        }

        return $response->getBody();
    }
    // ...
}
```

Why is this class difficult to unit test?

What if we want unit tests to run fast without waiting for the network?

What if we want unit tests to run fast without logging?

```php
<?php
use Guzzle\Http\Client;
use Acme\Logger\XmlLogger;

class FeedAggregator {
    private $client;
    private $logger;

    public function __construct () {
        $this->client = new Client();
        $this->logger = new XmlLogger();
    }

    public function retrieveFeed ($host, $path) {
        $request = $this->client
        $response = $request->se
        if (200 != $response->ge
            $this->logger->log('Coul
            return null;
        }

        return $response->getBody();
    }
    // ...
}
```

What if we ever want to use a different HTTP client?

Why is this class difficult to unit test?

What if we want unit tests to run fast without waiting for the network?

What if we want unit tests to run fast without logging?

```php
<?php
use Guzzle\Http\Client;
use Acme\Logger\XmlLogger;

class FeedAggregator {
    private $client;
    private $logger;

    function __construct () {
        $this->client = new Client();
        $this->logger = new XmlLogger();
    }

    public function retrieveFeed ($host, $path) {
        $request = $this->client->get($host.$path);
        $response = $request->send();
        if (200 != $response->getStatusCode()) {
            $this->logger->log('Could not get: '.$host.$path);
            return null;
        }

        return $response->getBody();
    }
    // ...
}
```

What if we ever want to use a different HTTP client?

Why is this class difficult to unit test?

What if we ever want to use a different logger class?

What if we want unit tests to run fast without waiting for the network?

What if we want unit tests to run fast without logging?

```php
<?php
use Guzzle\Http\Client;
use Acme\Logger\XmlLogger;

class Fe
    private $client;
    private $logger;

    public function __construct()
    {
        $this->client = new Client();
        $this->logger = new XmlLogger();
    }

    public function retrieveFeed($host, $path)
    {
        $request = $this->client->get($host.$path);
        $response = $request->send();
        if (200 != $response->getStatusCode()) {
            $this->logger->log('Could not get: '.$host.$path);
            return null;
        }

        return $response->getBody();
    }
    // ...
}
```

What if we ever want to use a different HTTP client?

Why is this class difficult to unit test?

What if we ever want to use a different logger class?

What if we ever want to use a different log format?

What if we want unit tests to run fast without waiting for the network?

What if we want unit tests to run fast without logging?

```php
<?php
use Guzzle\Http\Client;
use Acme\Logger\XmlLogger;

class Fo
    }

    public func
        $request
        $response = $request->se
        if (200 != $response->ge
            $this->logger->log('Coul
            return null;
        }

        return $response->getBody();
    }
    // ...
}
```

What if we ever want to use a different HTTP client?

Why is this class difficult to unit test?

What if we ever want to use a different logger class?

What if we ever want to

What if we want unit tests to run fast aiting for the network?

Dependencies are **pulled**.
=> Replacing requires refactoring
=> Dynamic replacing (only for testing) is **impossible**

What if we want unit tests to run fast without logging?

```php
<?php
use    Guzzle\Http\Client;
use    Acme\Logger\XmlLogger;

class FeedAggregator {
    private $client;
    private $logger;

    public function __construct () {

        $this->client = new Client();
        $this->logger = new XmlLogger();
    }

    public function retrieveFeed ($baseurl, $path) {
        $request = $this->client->setBaseUrl($baseurl)->get($path);
        $response = $request->send();
        if (200 != $response->getStatusCode()) {
            $this->logger->log('Could not get: '.$host.$path);
            return null;
        }

        return $response->getBody();
    }
    // ...
}
```

Dependencies are **pulled**.

Dependencies are **pushed**.

```php
<?php
use   Guzzle\Http\ClientInterface;
use   Acme\Logger\LoggerInterface;

class FeedAggregator {
    private $client;
    private $logger;

    public function __construct(ClientInterface $client,
                                LoggerInterface $logger) {
        $this->client = $client;
        $this->logger = $logger;
    }

    public function retrieveFeed ($baseurl, $path) {
        $request = $this->client->setBaseUrl($baseurl)->get($path);
        $response = $request->send();
        if (200 != $response->getStatusCode()) {
            $this->logger->log('Could not get: '.$host.$path);
            return null;
        }

        return $response->getBody();
    }
    // ...
}
```

```php
<?php
use   Guzzle\Http\ClientInterface;
use   Acme\Logger\LoggerInterface;

class FeedAggregater
    private $client
    private $logger

    public function __construct(ClientInterface $client,
                                LoggerInterface $logger) {
        $this->client = $client;
        $this->logger = $logger;
    }

    public function retrieveFeed ($baseurl, $path) {
        $request = $this->client->setBaseUrl($baseurl)->get($path);
        $response = $request->send();
        if (200 != $response->getStatusCode()) {
            $this->logger->log('Could not get: '.$host.$path);
            return null;
        }

        return $response->getBody();
    }
    // ...
}
```

Dependencies are **pushed**.

Class only depends on **interfaces**

```php
<?php
use   Guzzle\Http\ClientInterface;
use   Acme\Logger\LoggerInterface;

class FeedAggregater
    private $client
    private $logger

    public function __construct(ClientInterface $client,
                                LoggerInterface $logger) {
        $this->client = $client;
        $this->logger = $logger;
    }

    public function retrieveFeed ($baseurl, $path) {
        $request = $this->client->setBaseUrl($baseurl)->get($path);
        $response = $request->send();
        if (200 != $response->getStatusCode()) {
            $this->logger->log('Could not get: '.$host.$path);
            return null;
        }

        return $response->getBody();
    }
    // ...
}
```

Dependencies are **pushed**.

Class only depends on **interfaces**

Implementations are **injected** at runtime

```php
<?php
use   Guzzle\Http\ClientInterface;
use   Acme\Logger\LoggerInterface;

class FeedAggregater
    private $client
    private $logger

    public function __construct(ClientInterface $client,
                                LoggerInterface $logger) {
        $this->client = $client;
        $this->logger = $logger;
    }

    public function retrieveFeed ($baseurl, $path) {
        $request = $this->client->setBaseUrl($baseurl)->get($path);
        $response = $request->send();
        if (200 != $response->getStatusCode()) {
            $this->logger->log('Could not get: '.$host.$path);
            return null;
        }

        return $response->getBody();
    }
    // ...
}
```

Dependencies are **pushed**.

Class only depends on **interfaces**

Implementations are **injected** at runtime

Easy to **replace**, even **dynamically** (for testing)

```php
<?php
use   Guzzle\Http\ClientInterface;
use   Acme\Logger\LoggerInterface;

class FeedAggregater
    private $client
    private $logger

    public function __construct(ClientInterface $client,
                                LoggerInterface $logger) {
        $this->client = $client;
        $this->logger = $logger;
    }

    public function retrieveFeed ($baseurl, $path) {
        $request = $this->client->setBaseUrl($baseurl)->get($path);
        $response = $
        if (200 != $re                                    {
            $this->log                                .$host.$path);
            return nul
        }

        return $response->getBody();
    }
    // ...
}
```

Dependencies are **pushed**.

Class only depends on **interfaces**

Implementations are **injected** at runtime

Easy to **replace**, even **dynamically** (for testing)

On the level of the class, You are now experts for **Dependency Injection**.

```php
<?php
use   Guzzle\Http\ClientInterface;
use   Acme\Logger\LoggerInterface;

class FeedAggregater
    private $client
    private $logger

    public function __construct(ClientInterface $client,
                                LoggerInterface $logger) {
        $this->client = $client;
        $this->logger = $logger;
    }

    public function retrieveFeed ($baseurl, $path) {
        $request = $this->client->setBaseUrl($baseurl)->get($path);
        $response = $
        if (200 != $re                                    {
            $this->log                                .$host.$path);
            return nul
        }

        return $response->getBody();
    }
    // ...
}
```

Dependencies are **pushed**.

Class only depends on **interfaces**

Implementations are **injected** at runtime

Easy to **replace**, even **dynamically** (for testing)

On the level of the class, You are now experts for **Dependency Injection**.

Any questions?

```php
<?php
use    Guzzle\Http\ClientInterface;
use    Acme\Logger\LoggerInterface;

class FeedAggregater
    private $client
    private $logger

    public function __construct(ClientInterface $client,
                                LoggerInterface $logger) {
        $this->client = $client;
        $this->logger = $logger;
    }

    public function retrieveFeed ($baseurl, $path) {
        $request = $this->client->setBaseUrl($baseurl)->get($path);
        $response = $
        if (200 != $re                                    {
            $this->log                                .$host.$path);
            return nul
        }

        return $response->getBody();
    }
    // ...
}
```

Dependencies are **pushed**.

Class only depends on **interfaces**

Implementations are **injected** at runtime

Easy to **replace**, even **dynamically** (for testing)

On the level of the class, You are now experts for **Dependency Injection**.

Who constructs and pushes all the dependencies?

Dependency Injection Container

"DI Container", "DIC", "Service Container", "the Container"

C++ [Bearbeiten]

- PocoCapsule/C++ IoC und DSM Framework

Java [Bearbeiten]

- Contexts and Dependency Injection (CDI), Standard für DI (JSR 299,[1] eine Rahmenrichtlinie, umgesetzt durch verschiedene Frameworks wie z. B. *Seam Weld* in Java EE 6)
- EJB ab Version 3.0
- Spring
- PicoContainer
- Seam 2
- Guice
- simject
- JBoss Microcontainer ab JBoss Application Server 5.0
- OSGi Declarative Services

PHP 5 [Bearbeiten]

- Garden (wird nicht mehr weiterentwickelt)
- Stubbles IoC
- Enterprise-PHP-Framework
- Symfony Components (BETA), Opensource PHP Standalone Classes
- Symfony2, Open-Source PHP Framework
- FLOW3, Open-Source PHP Framework
- Phemto
- PicoContainer for PHP
- Pimple
- pinjector
- Zend Framework 2, Opensource PHP Framework
- Adventure PHP Framework

Perl [Bearbeiten]

- Bread::Board
- Orochi

Ruby [Bearbeiten]

- Copland
- Needle

Python [Bearbeiten]

- PyContainer
- SpringPython
- snake-guice
- python-inject

.NET [Bearbeiten]

- Autofac
- Ninject
- Spring.NET
- Structuremap
- Unity Application Block
- Puzzle.NFactory
- Castle MicroKernel und Windsor Container
- NauckIT.MicroKernel
- Managed Extensibility Framework
- ObjectBuilder
- PicoContainer.NET
- WINTER4NET
- LightCore
- OpenNETCF.IoC
- LOOM.NET mit Dependency Injection Aspect
- PRISM

ColdFusion [Bearbeiten]

- ColdSpring
- LightWire

Actionscript [Bearbeiten]

- Swiz
- Parsley
- Cairngorm 3
- Robotlegs
- StarlingMVC

Objective C [Bearbeiten]

- Objection

Delphi [Bearbeiten]

- Spring Framework for Delphi

Very Many Frameworks support Dependency Injection

C++ [Bearbeiten]

- PocoCapsule/C++ IoC und DSM Framework

Java [Bearbeiten]

- Contexts and Dependency Injection (CDI), Standard für DI (JSR 299,[1] eine Rahmenrichtlinie, umgesetzt durch verschiedene Frameworks wie z. B. *Seam Weld* in Java EE 6)
- EJB a
- Spring
- PicoC
- Seam
- Guice
- simject
- JBoss Microcontainer ab JBoss Application Server 5.0
- OSGi Declarative Services

PHP 5 [Bearbeiten]

- Garden (wird nicht mehr weiterentwickelt)
- Stubbles IoC
- Enterprise-PHP-Framework
- Symfony Components (BETA), Opensource PHP Standalone Classes
- Symfony2, Open-Source PHP Framework
- FLOW3, Open-Source PHP Framework
- Phemto
- PicoContainer for PHP
- Pimple
- pinjector
- Zend Framework 2, Opensource PHP Framework
- Adventure PHP Framework

Perl [Bearbeiten]

- Bread::Board
- Orochi

Ruby [Bearbeiten]

- Copland
- Needle

ColdFusion [Bearbeiten]

- ColdSpring

.NET [Bearbeiten]

- Autofac
- Ninject
- Spring.NET
- Structuremap
- Unity Application Block
- Puzzle.NFactory
- Castle MicroKernel und Windsor Container
- NauckIT.MicroKernel
- Managed Extensibility Framework
- ObjectBuilder
- PicoContainer.NET
- WINTER4NET
- LightCore
- OpenNETCF.IoC
- LOOM.NET mit Dependency Injection Aspect
- PRISM

- Robotlegs
- StarlingMVC

Objective C [Bearbeiten]

- Objection

Delphi [Bearbeiten]

- Spring Framework for Delphi

Vielen Dank
für Eure Aufmerksamkeit!

SOLID

- S Single Responsibility Principle
- O Open / Close Principle
- L Liskov Substitution Principle
- I Interface Segregation Principle
- D Dependency Inversion Principle