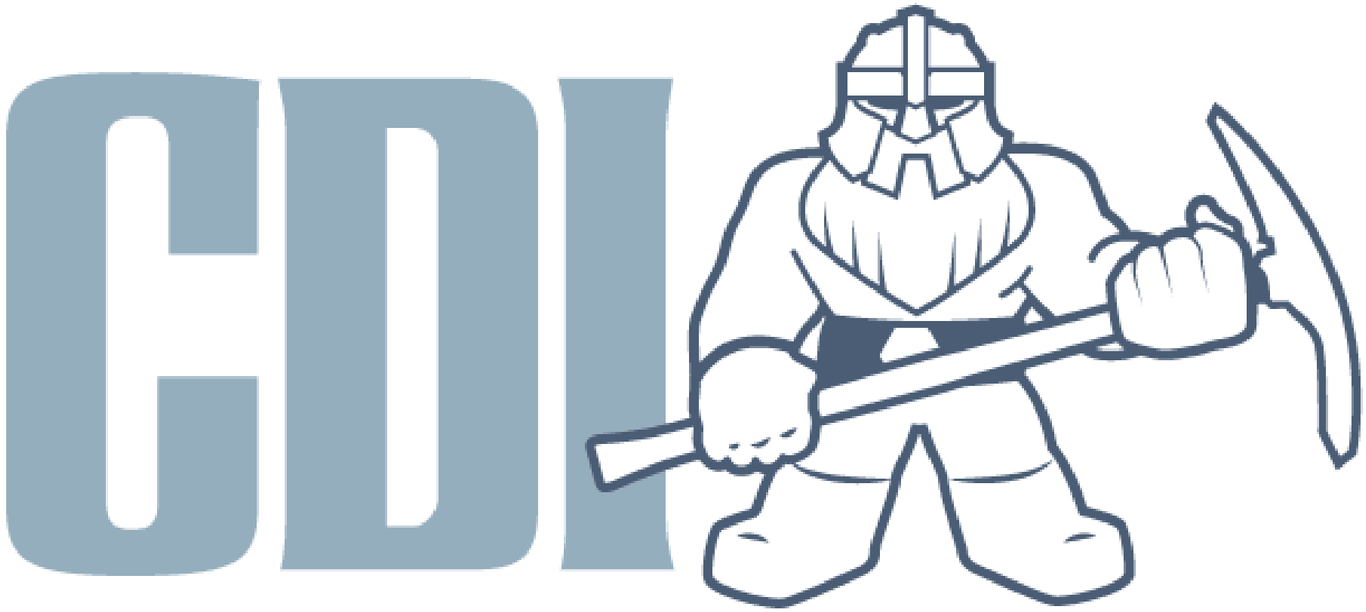
JavaOne™

CDI 2.0

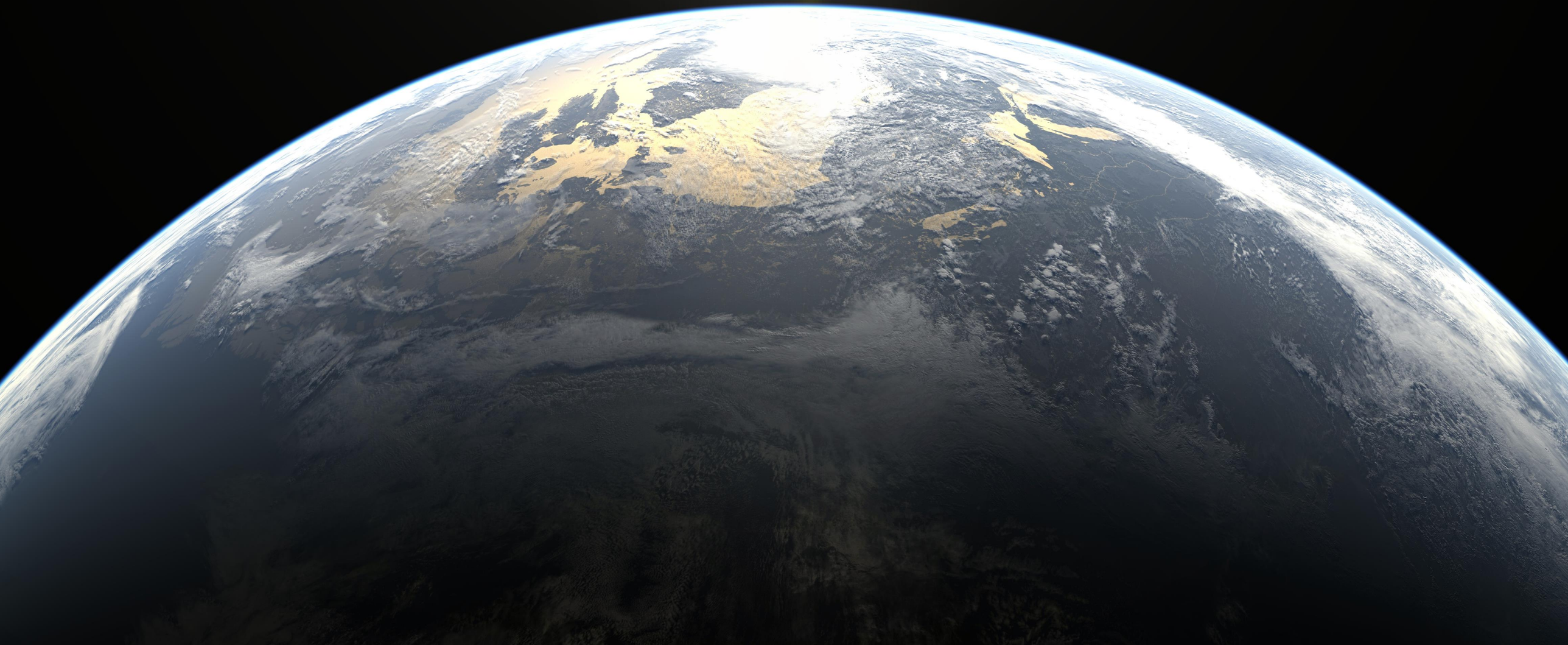
what's in the work?



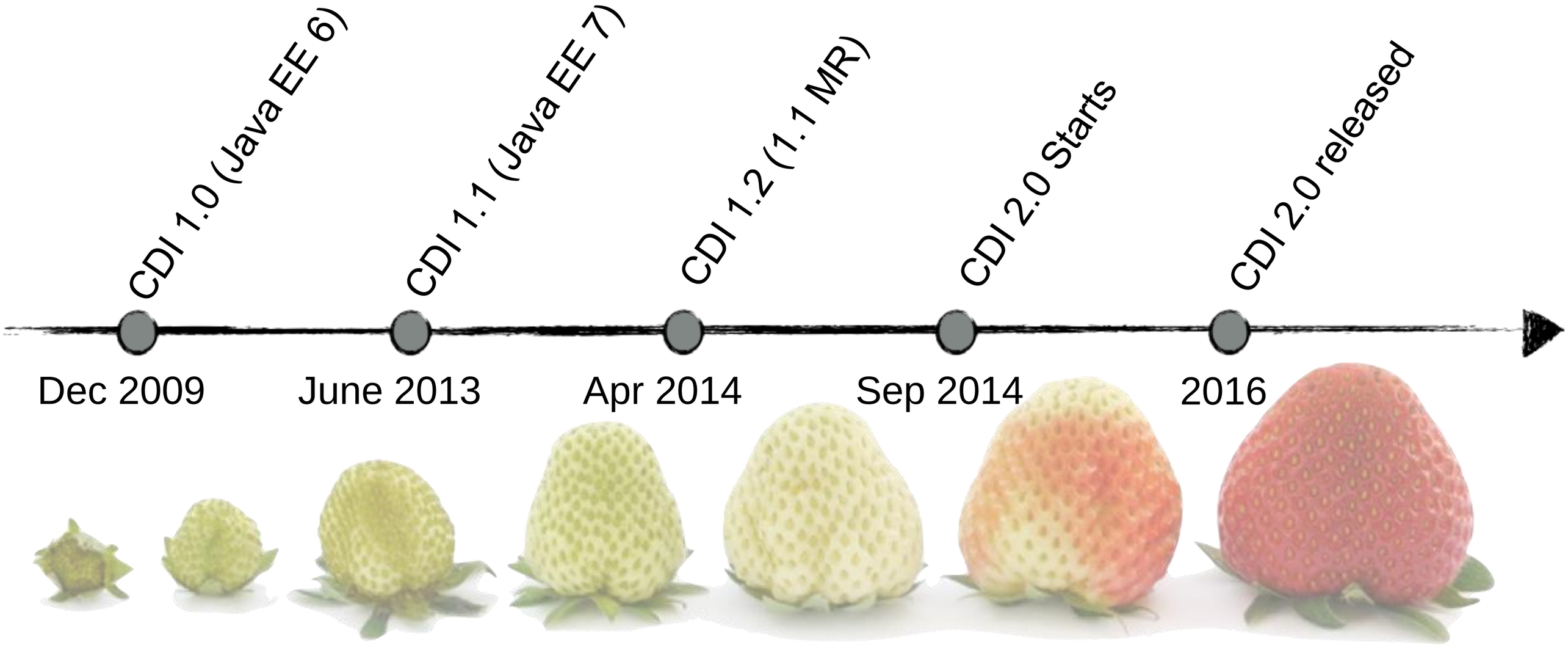
Agenda

- Flashback on CDI 1.0, 1.1 and 1.2
- CDI 2.0 status
- Gathering feedback for CDI 2.0
- CDI 2.0 new features
- Questions and Feedback

Previously on CDI



CDI Timeline



CDI 1.0 – December 2009

- A typesafe dependency injection mechanism
- A well-defined lifecycle for stateful objects
- The ability to decorate or to associate interceptors to objects with a typesafe approach
- An event notification model
- An SPI allowing portable extensions

CDI 1.1 – June 2013

- CDI is automatically enabled in Java EE
- Introspection with bean, events, decorator and interceptor metadata
- Ease access to CDI from non CDI code
- Work on interceptor for reuse by other Java EE specs
- SPI enhancement for portable extensions

CDI 1.2 – April 2014

- Clarifications in the spec
  - Lifecycles
  - Events
  - Conversation scope
- Fix conflict with other JSR 330 frameworks
- OSGi support in the API

CDI 2.0 started 12 months ago

- JSR 365!
  - First Java EE 8 JSR proposed and voted
- Weekly IRC meeting
- Regular release of Weld 3.0 Alpha (CDI 2.0 RI)
- We have a lot of community momentum
- Early Draft is around the corner
- Release expected in 2016 (Q2?)

EG members

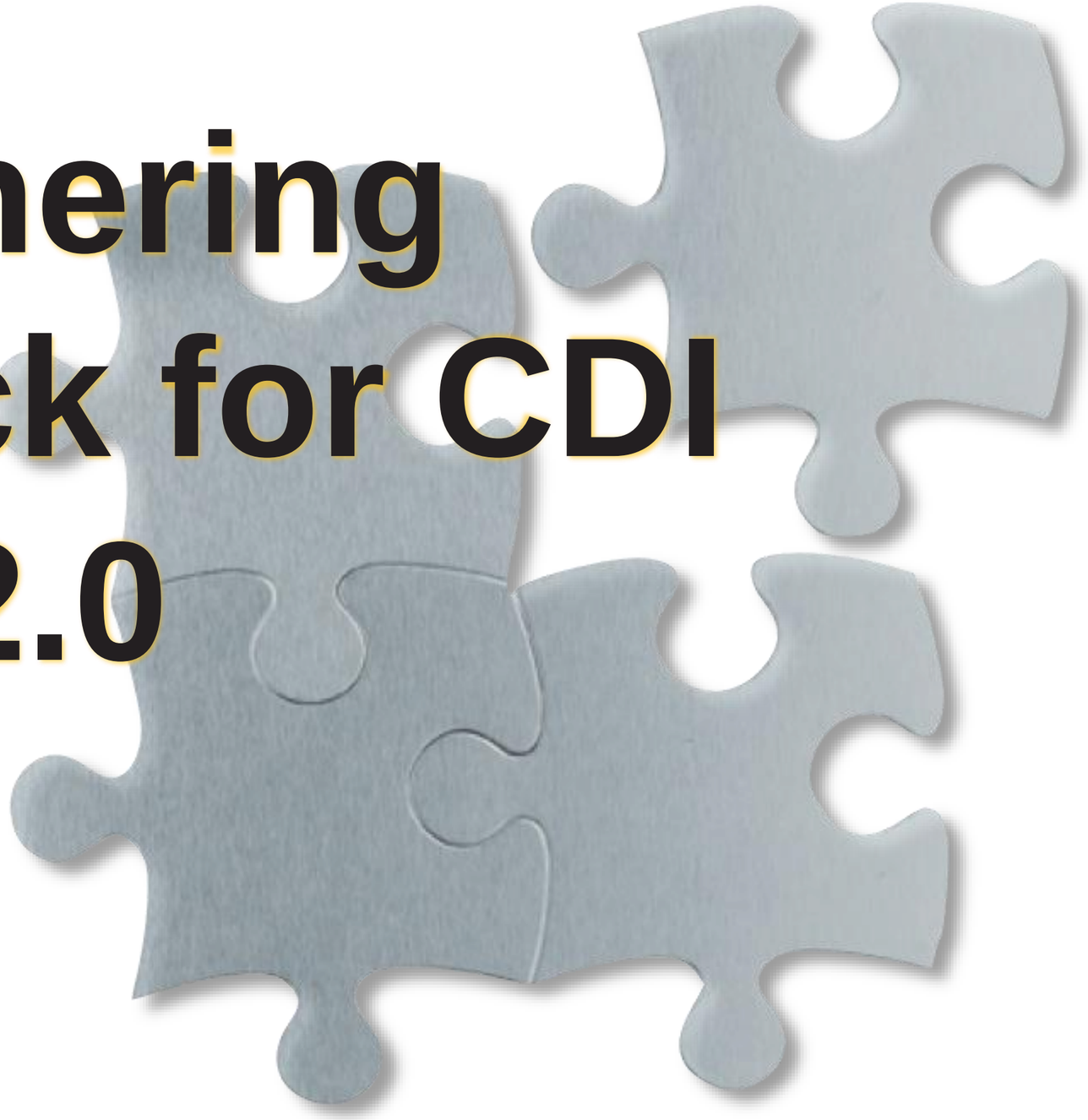
- **Pete Muir (Red Hat)**
- **Antoine Sabot-Durand (Red Hat)**
- José Paumard
- John Ament
- David Currie (IBM)
- Anatole Tresch (Credit Suisse)
- Antonio Goncalves
- Thorben Janssen
- Raj. Hegde (JUG Chennai)
- Werner Keil
- Joseph Snyder (Oracle)
- Mark Paluch
- Florent Benoit (SERLI)
- Mark Struberg
- David Blevins (Tomitribe)
- George Gastaldi (Red Hat)
- Otavio Santana

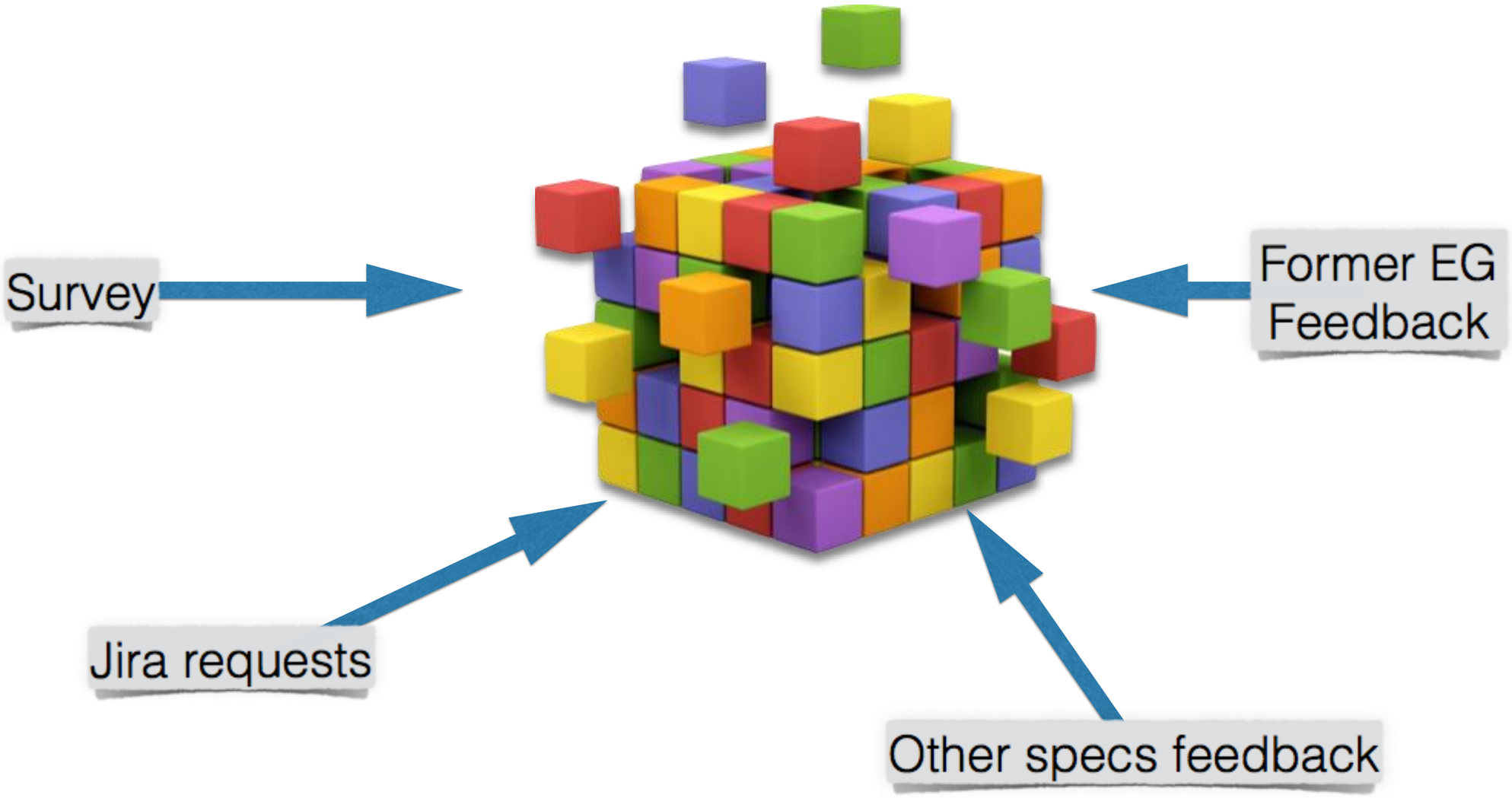
We are open to the community!

Everybody can participate by joining the mailing list :
https://lists.jboss.org/mailman/listinfo/cdi-dev

More information on our website :
http://cdi-spec.org

Gathering feedback for CDI 2.0

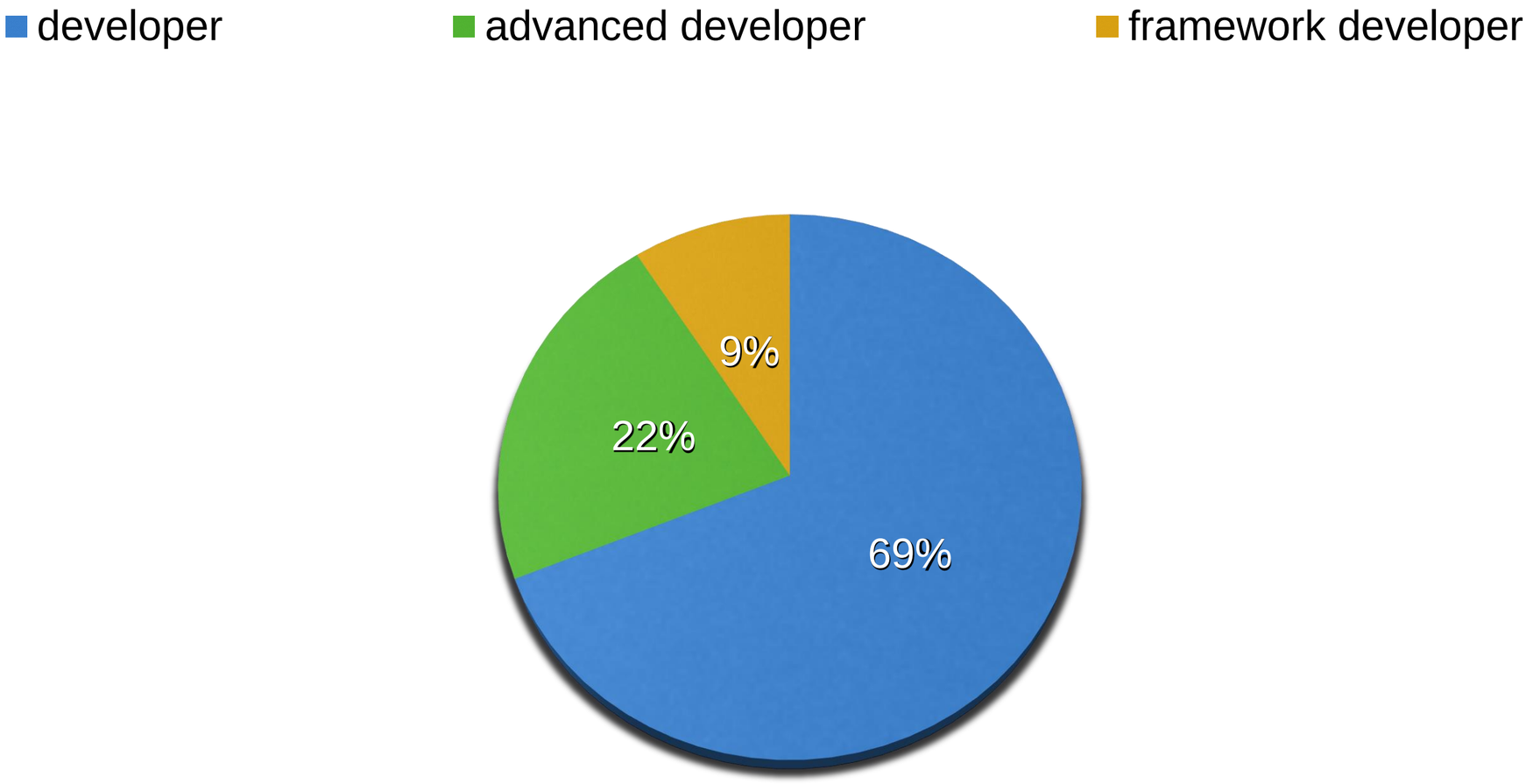



Survey
Former EG Feedback
Jira requests
Other specs feedback

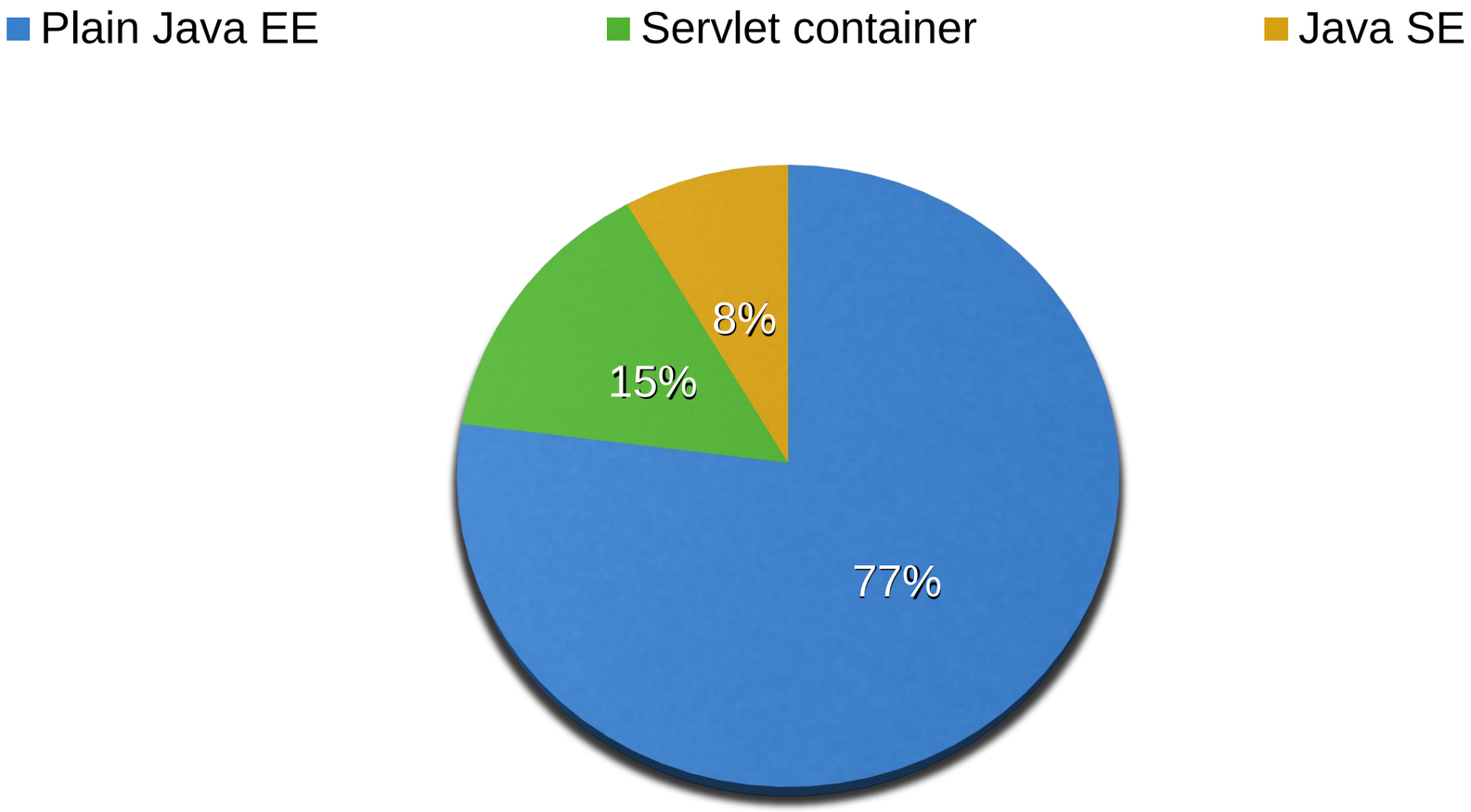
CDI 2.0 survey

260 participants
20 features to rate

Who answered?



Who answered?



1st feature

- Asynchronous support
  for events and method invocation

Other top requested features

- @Startup for CDI
- Bootstraping outside of Java EE
- AOP for custom beans
- Security support
- Observers ordering, better event control
- Access to metadata through SPI

CDI 2.0 new features

Java SE support

Using CDI outside of the Java EE Container

Why that?

- To ease the testing of CDI applications
- To provide a mean of building new stacks out of Java EE
- To boost CDI adoption for Spec working already on Java SE

- First step before working on a CDI light

Java SE support will start in EDR1

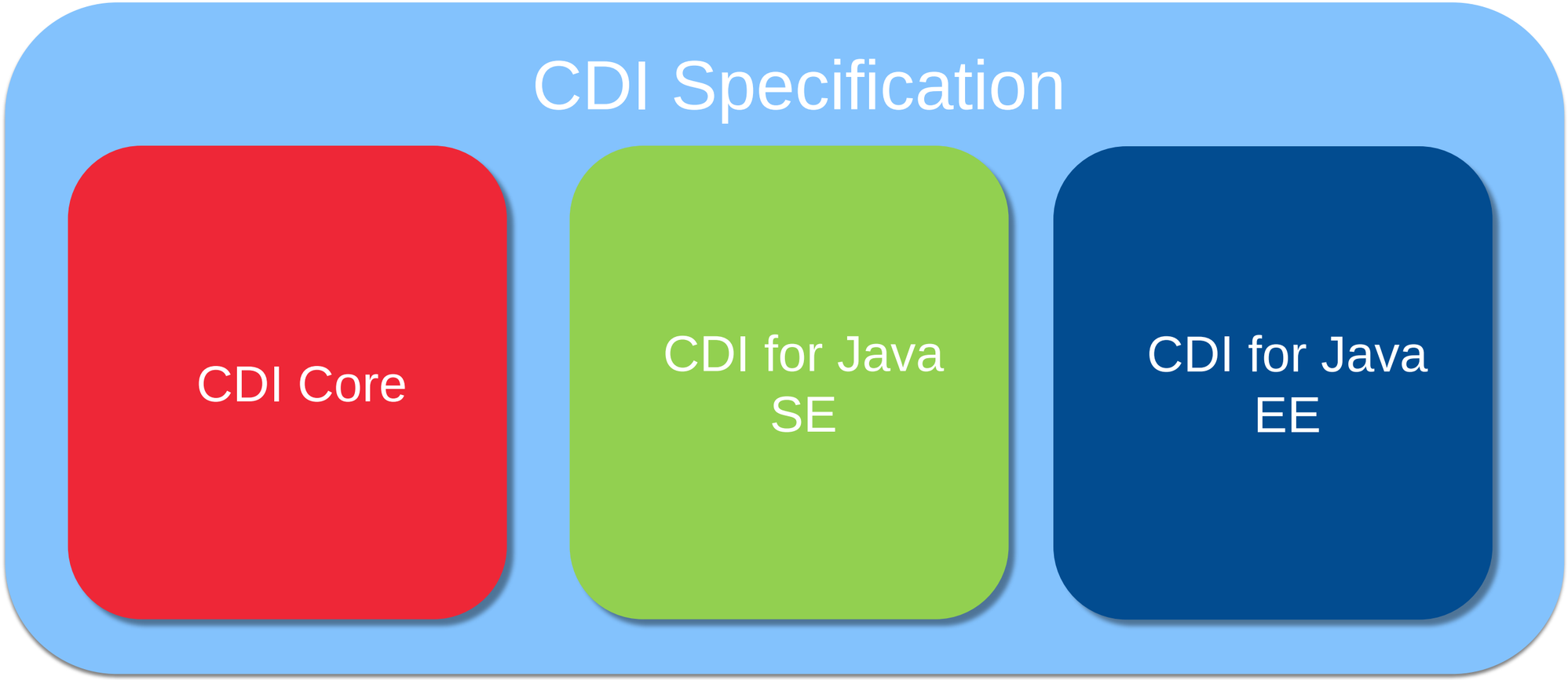
- We specified API to boot CDI in Java SE:

```
public static void main(String... args) {

    CDIProvider provider = CDI.getCDIProvider();
    CDI<Object> cdi = provider.initialize();
    // retrieve a bean and do work with it
    MyBean myBean = cdi.select(MyBean.class).get();
    myBean.doWork();
    // when done
    cdi.shutdown();
}
```

- Desktop and non Java EE application can now use a standard way to boot CDI

What did we do?



There’s still work to do

- What about built-in contexts activation in Java SE?
  - RequestScope
  - SessionScope
  - ConversationScope

There’s still work to do

- What about bean discovery in Java SE?
- Annotated mode can be very costly
- Implicit bean archive even more (support is disable now)

- What about support of multiple container in Java SE?

Modularity

Provide sub specs in CDI (called parts)
that can be used independently
Each part should have an implementation

Why that?

- To avoid the “bloated spec” syndrom
- Having parts will help CDI adoption
- Third party won’t have to implement the whole spec if they don’t want to

Modularity – 2 core parts

Full CDI

CDI Light

- Basic DI
- Producers
- Programmatic lookup
- Singleton and dependent scopes
- Basic SPI for integration

- Events
- Normal scopes
- Interceptor & Decorator
- Advanced SPI

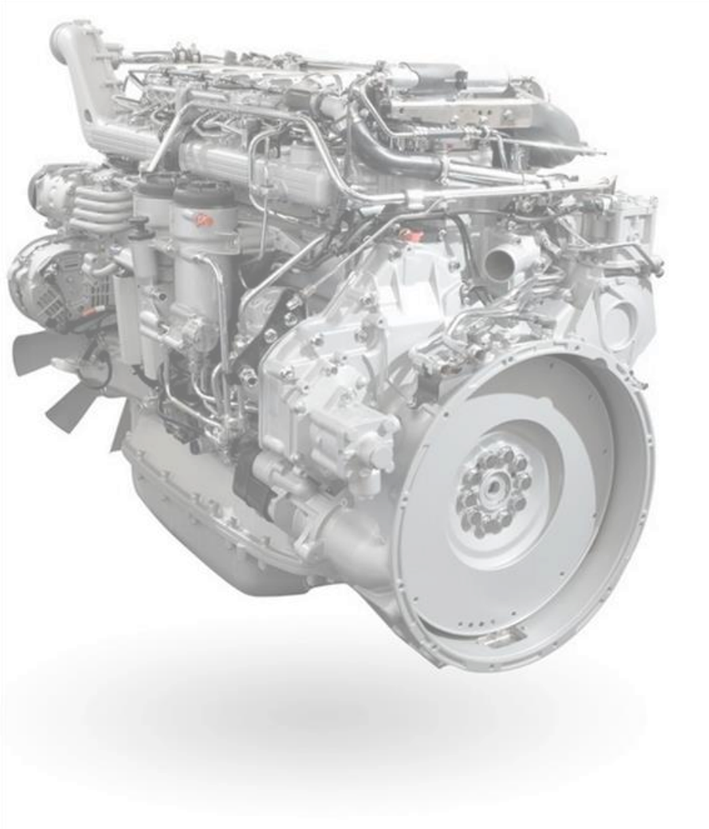
Modularity – challenges

- Will bring 4 subspec:
  - CDI light for Java SE
  - CDI full for Java SE
  - CDI light for Java EE
  - CDI full for Java EE

- Having an RI and TCK for each part can be an important work

Enhancing events

Making a popular feature even more popular!



Enhancing Events

- CDI events are a very loved feature!

For CDI 2.0, we plan to introduce :

- Asynchronous events
- Events ordering

Events in CDI 1.x: patterns

- Firing pattern:

```
@Inject
Event<Payload> event;

public void someCriticalBusinessMethod() {

    event.fire(new Payload());
}
```

Events in CDI 1.x: patterns

- Observing pattern:

```
public void callMe(@Observes Payload payload) {

    // Do something with the event
}
```

- Supports qualifiers and many other things

CDI 1.x: Sync / Async

- Sync / Async is not specified
- The immutable status of the payload is not specified

- Implementations use a Sync model
- The payload is mutated in some implementations / framework

- Going async “blindly” might raise problems…

Events are sync in CDI 1

Right now:

- All the observers are called in the firing thread
- In no particular order (at least not specified)
- The payload may be mutated

Events and contexts

Contexts

- Two contexts are critical: transactions and HTTP requests / sessions
- Events are aware of those contexts

- In an all-sync world, everything is fine
- But in an async world, we will be in trouble

Asynchronous Events

- So designing backward compatible async events is more tricky than it looks:

1) A currently sync event should remain sync
2) Going sync / async should be a decision taken from the firing side
3) Being sync should be possible from the observing side

Asynchronous Events

- Pattern for the firing side:

```
@Inject
Event<Payload> event;

public void someOtherCriticalBusinessMethod() {

    event.fireAsync(new Payload());
}
```

Asynchronous Events

- Pattern for the observing side:

```
public void callMe(@Observes Payload payload) {

    // I am called in the firing thread
    // Whether is was async fired or not
}
```

```
public void callMe(@ObservesAsync Payload payload) {

    // I am called in another thread
}
```

Asynchronous Events

- So, in a nutshell

	`callMe(@Observes payload)`	`callMe(@ObservesAsync payload)`
`event.fire(payload)`	Sync call	Not notified
`event.fireAsync(payload)`	Not notified	Async call

What about mutable payloads?

- One short answer:
- Don’t do it!
- Or suffer the full penalty of race conditions!

We have some more

- Let us come back to this pattern:

```
@Inject
Event<Payload> event;

public void someOtherCriticalBusinessMethod() {

    event.fireAsync(new Payload());
}
```

- 1st question: in what thread are the observers going to be called?

We have some more

- Let us come back to this pattern:

```
@Inject
Event<Payload> event;

public void someOtherCriticalBusinessMethod() {

    event.fireAsync(new Payload());
}
```

- 2nd question: what if exceptions are thrown by the observers?

Adding an Executor to fireAsync

- What if the observer needs to be called in the GUI thread?

```
@Inject
Event<PanelUpdater> event;

public void someOtherCriticalBusinessMethod() {

    event.fireAsync(new PanelUpdater(green),
                    executor); // Of type Executor
}
```

Adding an Executor to fireAsync

- What if the observer needs to be called in the GUI thread?

```
@Inject
Event<PanelUpdater> event;

public void someOtherCriticalBusinessMethod() {

    event.fireAsync(new PanelUpdater(green),
                    SwingUtilities::invokeLater);
}
```

Handling exceptions

- The firing async is built on the Java 8 async model: CompletionStage

```
@Inject
Event<PanelUpdater> event;

public void someOtherCriticalBusinessMethod() {

    CompletionStage<PanelUpdater> stage =
        event.fireAsync(new PanelUpdater(green),
                        SwingUtilities::invokeLater);
}
```

Handling exceptions

- Two ways of handling exceptions:

```
stage.exceptionaly( // Function
    exception -> doSomethingWith(exception));
```

Returns a new CompletionStage

- That completes when the CS completes
- Either with the same result (normal completion)
- Or with the transformed exception

Handling exceptions

- Two ways of handling exceptions:

```
stage.handle( // BiFunction
    (result, exception) -> doSomethingWith(result, exception));
```

Returns a new CompletionStage

- That completes when the CS completes
- Calls the BiFunction with a null as result or exception

- As a bonus: observers can return objects!

Handling exceptions

- Two ways of handling exceptions:

```
stage.handle( // BiFunction
    (result, exception) -> doSomethingWith(result, exception));
```

- The returned exception is a FireAsyncException
- It holds all the exceptions in the suppressed exception set

Events ordering

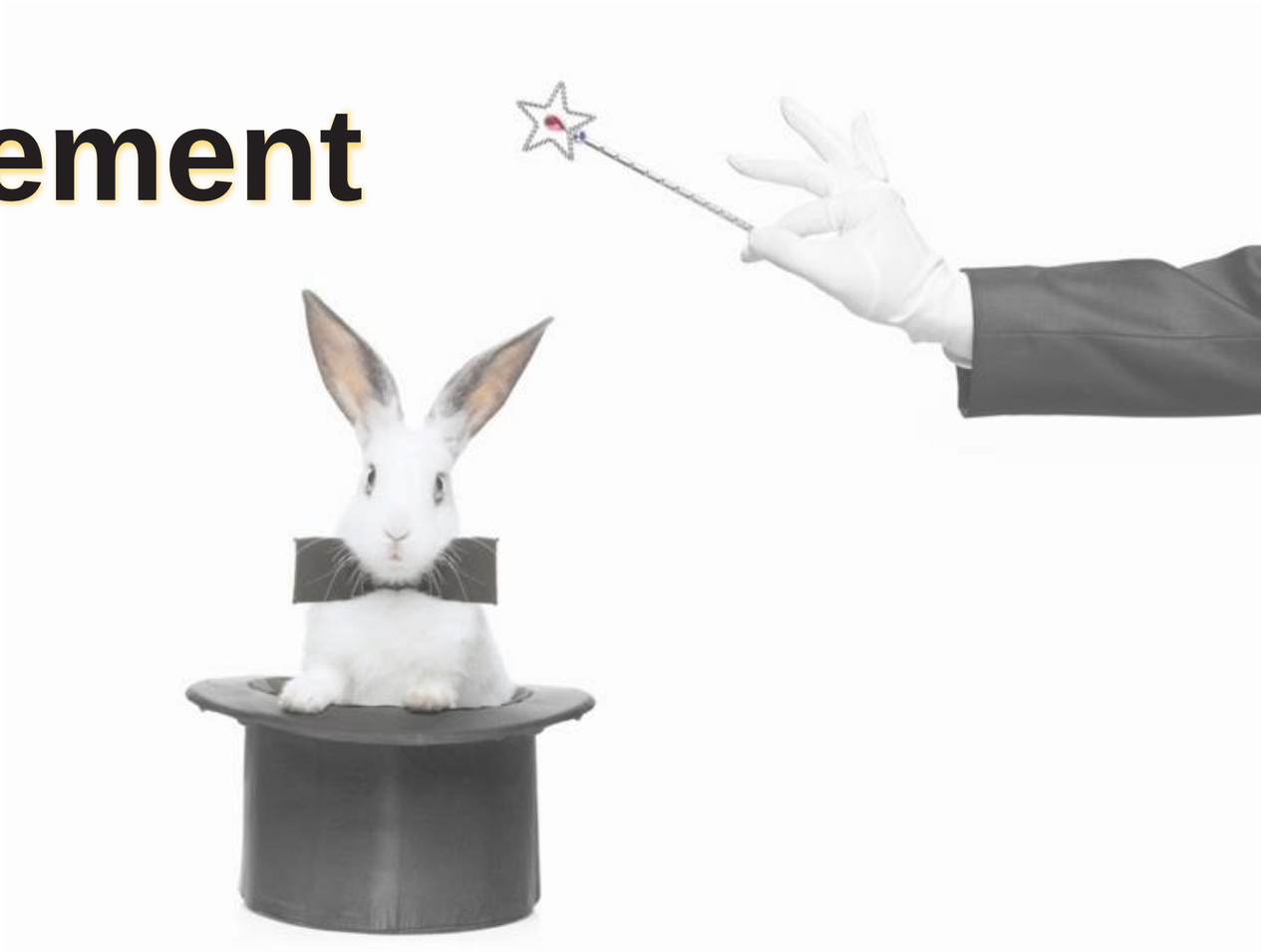
- Pattern:

```
public void firstObserver(@Observes @Priority(1) Payload p) {}

public void secondObserver(@Observes @Priority(2) Payload p) {}
```

- Ordering in async… possible but complex

AOP Enhancement



Support AOP on producer

- In CDI 1.x you cannot bind an interceptor to a produced bean

```
@Produces
@Transactional
public MyService produceService() {

   ...
}
```

- `@Transactional` is applied to producer method

Solution: BeanInstanceBuilder

```
public class MyAdvancedProducerBean {

    public BeanInstanceBuilder<MyClass> bib = new BeanInstanceBuilder<>();

    @Produces
    @RequestScoped
    public MyClass produceTransactionalMyClass() {
        AnnotatedTypeBuilder<MyClass> atb = new AnnotatedTypeBuilder<>()
                .readFrom(MyClass.class)
                .addToMethod(MyClass.class.getMethod("performInTransaction")
                        , new TransactionalLiteral());

        return bib.readFromType(atb.build())
                .build(); //instance of the bean with requested interceptors / decorators
    }

    public void disposeMyClass (@Disposes Myclass td) {
        bib.dispose(td);
    }
}
```

CDI 2.0 DEMO

CDI 2.0 needs you!!

CDI 2.0 specification is open to everyone
Mailing list, IRC channel

http://cdi-spec.org @cdispec


Which JSR
you'll use
365 days a year?
JSR 365!!

Q & A