Adopting the Eclipse Communication Framework: The Case of eConference

Fabio Calefato, **Mario Scalas**
Università degli Studi di Bari
Dipartimento di Informatica

Outline

- About eConference
- Adopting ECF
  - Motivation
  - Cost of change
- Architectural blocks
  - Shared Object Proxy
  - Dependency Injection
- Conclusion and future work

About eConference

- A text-based conferencing tool supporting ad-hoc distributed teams
- Built on RCP from Release 3
- We could deliver modularized features as plugins (Co-browsing web, PDF, ...)
- Release 4 has a new architecture and integrates Eclipse Communication Framework

Adopting ECF: Motivations /1

- Network infrastructure in eConference 3 was too expensive to maintain for our organization
  - New capabilities had to be provided by implementing them in XMPP on our own
- ECF is a framework for supporting the development of distributed Eclipse-based tools and applications
- Shared Objects API
  - A set of abstractions for sharing objects across the network

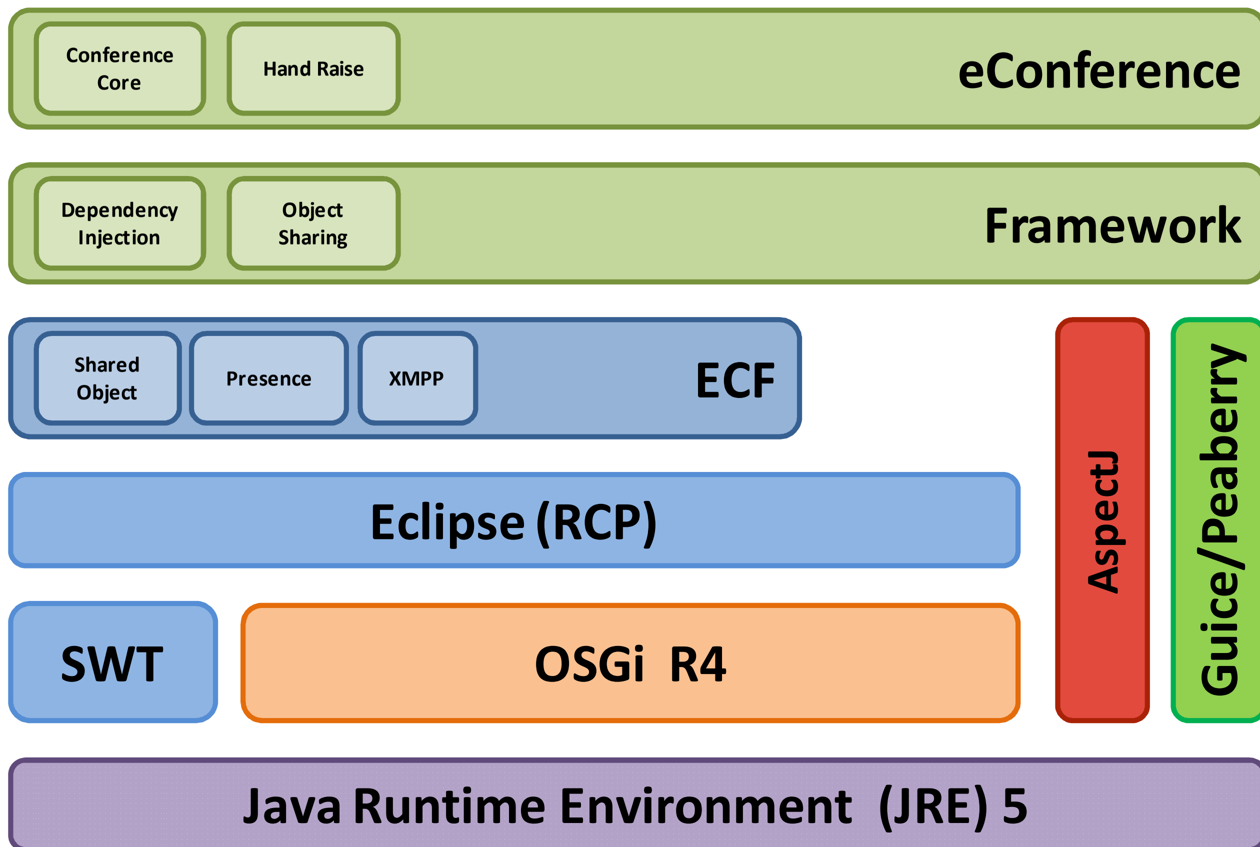
Adopting ECF: Motivations /2

- Comes with built-in network provider abstractions
  - XMPP, Skype, MSN, ...
- Has out-of-the-box ready to use basic components
  - contacts view, connection wizards, chat view, ...
- Has an active community

Adopting ECF: Cost

- Initial approach was to only rewrite the network layer by rebuilding it on ECF
  - The domain/business layer too coupled with our own framework
- Had to fall back on a full rewrite
  - “Reuse” of our experience to build a better design
  - Limited portions of existing code reused

Architectural Blocks



Main Design Enhancements

- ECF-based
  - Reuse backend infrastructure (API and events)
  - Reuse frontend widgets (contacts view, connection wizards, ...)
  - Make the API easier to use
- Dependency Injection
  - Never search for objects, just ask for them
  - Usen Java @nnotations

Making ECF API easier to use ...

Drawbacks of ECF

- Shared Objects API difficult to understand and handle
  - Lack of high level documentation
  - Need for tracing the communication protocol implementation to understand the API behaviour
- Highly multithreaded and asynchronous code
  - Tracing was hard

Use Proxy for Shared Objects

- ECF Shared Object API is clumsy
  - event subscriptions, checks, object serialization, ...
- We used the MVC pattern to implement plugins
  - Almost every plugin has a model that has to be shared
  - We observed that an implementation pattern was recurrent

How is our proxy implemented?

- We provided a more useful base class for “models to be shared”
  - **AbstractSharedObject**
- We introduced
  - **@MapProperty** to mark fields that are automatically replicated to remote clients
  - **@Replicate** to mark methods that must be locally- and remotely-executed

Shared Object Example /1

```
// An interface is required (and is a good practice)
public interface IMyModel {
  @Replicate void doSomething();
}

// Implement your model
public class MyModel extends AbstractSharedObject implements IMyModel {
  @MapProperty private String something;

  public void doSomething() { ... }
}
```

Shared Object Example /2

```
// Share your model with clients
private ISharedObjectProxy proxy;

public MyManager() {
  this.proxy = SharedObjectProxy.create( IMyModel.class );
}
...
// Construct & share a model (primary client) on a chat room
proxy.attach( getModelID(), chatRoomContainer );
proxy.share( new MyModel() );
…
//This will invoke the method on the local and remote clients
((IModel) proxy).doSomething();
```

Pro & Cons

- ☺ Easier to use than straight ECF API
- ☹ Client code must be proxy-aware
  - Using a proxy as a model is straighforward
  - The proxy life-cycle must be managed explicitly
  - Observer semantics is preserved but each client has to manage observers by its own
- **Is a better approach possible?**
  - Use AOP to hide this management behind the scenes

Decoupling objects

Why bother?

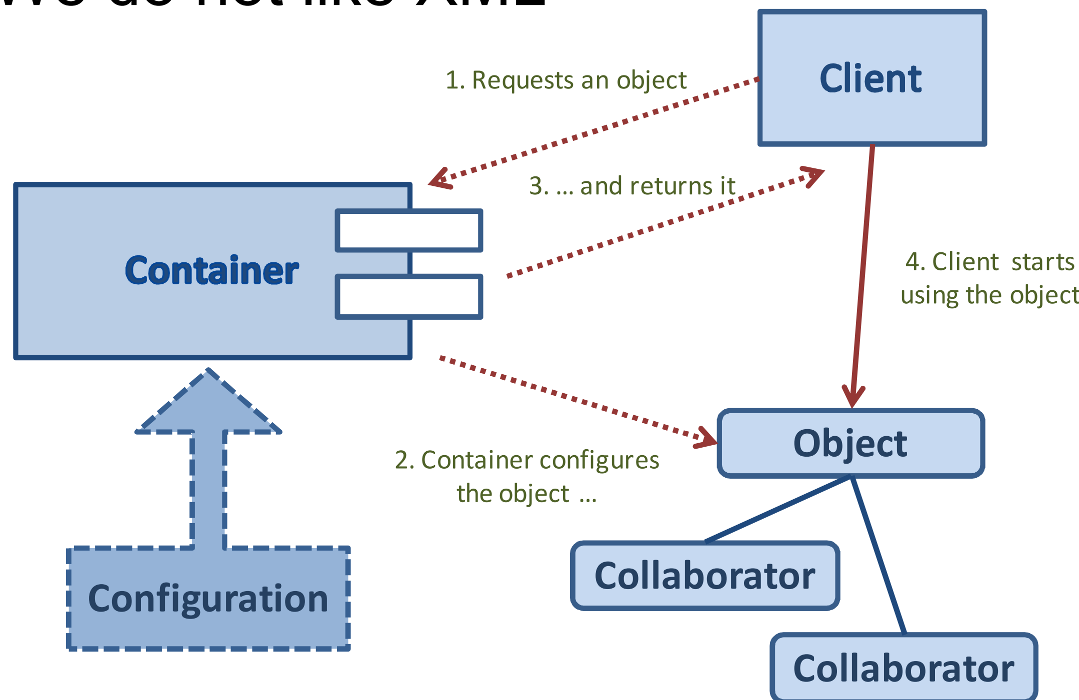
- Singletons are evil
  - As any other hard dependency
- Higly coupled OO code is difficult to test
  - Dependency Injection (DI) promotes composition of objects
- Object compositions are easier to test
  - We can provide test doubles to mock/stub real collaborators

Dependency Injection in pills

- Dependency injection is an answer to configuring objects
  - Objects renounce to search for their collaborators
- A **Container** will provide the collaborators
  - It must be instructed (configured) on how to find them
- We can provide different sets of configurations
  - One for production, one for testing, ...

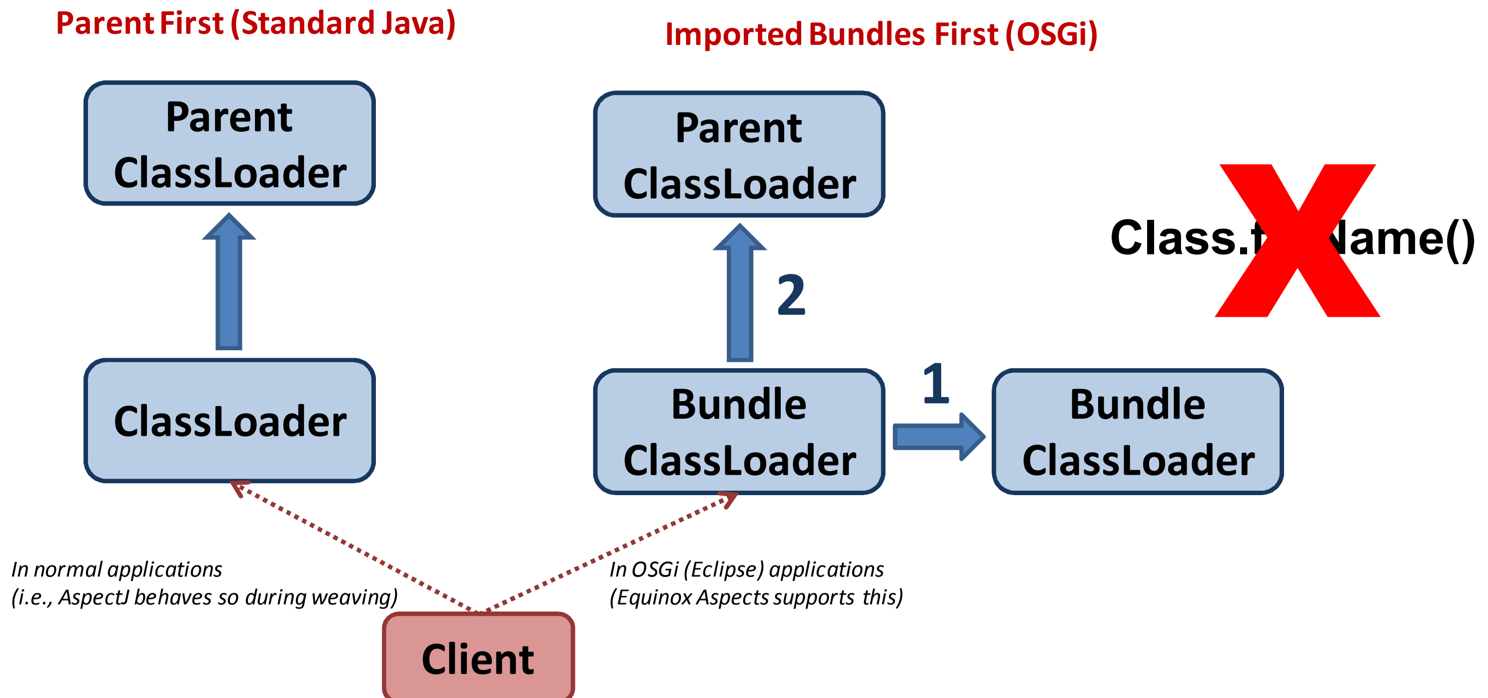
Configuration

- A container that can be configured in Java
  - We do not like XML



Classloading issues

- Support for non-delegating classloaders (OSGi)



AOP for Plugins

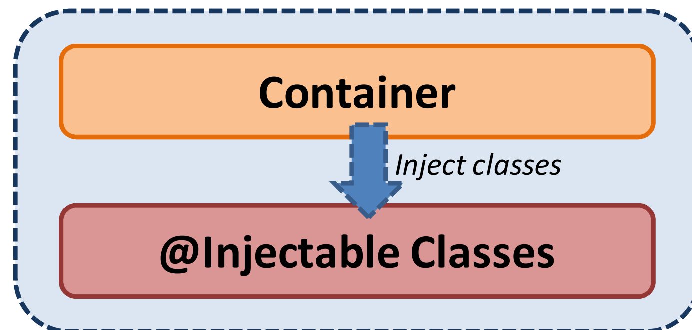
- AOP cleanly separates primary concerns (e.g., domain logic) from secondary concerns (e.g., thread safety)
- AspectJ doesn’t support OSGi
- Equinox Aspects is an Eclipse project to enhance AspectJ runtime to be OSGi-friendly
  - Aspects can get woven/unwoven as bundles dynamically come and go away

Dependency Injection Concern

- Dependency Injection can be seen as systemic concern regarding “object configuration”
- It is completely general
  - But we still need to cope with OSGi specifics
- Each bundle has its own configuration
  - An abstract base aspect provides all what is needed
  - Only define the packages that must be taken in consideration for injecting

Configuring Eclipse-created Stuff

- The container must allow to configure objects created externally (i.e., Eclipse workbench)
  - Views, commands, ...



Dependency Injection Aspect

Without AOP

```
public class ActionDelegate {
  @Inject ISomeService someService;
  public ActionDelegate() {
     Container.getInstance().inject( this );
  }
}
```

With AOP

```
@Injectable
public class ActionDelegate {
  @Inject ISomeService someService;
}
```

Pro & Cons

- ☺ Dependency Injection for OSGi services
  - Thanks to Guice/Peaberry
- ☹ Eclipse extensions support is missing
  - Must extend Guice with additional support
- ☺ No need for special tooling
  - Guice uses a Java DSL: any Java editor is ok
- ☹ Not a standard solution
  - Spring DM destined to rule the OSGi/Eclipse world

Conclusions

- We have the basic blocks for a more general application framework
  - We try to leverage existing technologies to make the our daily work simpler
  - Maybe it will be useful for somebody else
- For the future we want to provide
  - The infrastructure for supporting Presenter-First approach within RCP
  - Extend Guice/Peaberry to support Eclipse concepts and technologies

Acknowledgments

This work has been

supervised by Filippo Lanubile

and

supported by the 2006 Eclipse Innovation Award