JavaPolis 2003

PicoContainer

Inversion-of-Control made easy

Jon Tirsén & Aslak Hellesøy
ThoughtWorks


BeJUG

Jon Tirsén


Nanning


pico
container


damagecontrol


ThoughtWorks®
The art of heavy lifting.℠


PREVAYLER


codehaus
01100100011001010111001101110000011011110111010

BeJUG

Aslak Hellesøy

middlegen

Qpox

pico container

damagecontrol

ThoughtWorks®
The art of heavy lifting.℠

XDoclet
Attribute Oriented Programming

codehaus
01100100011001010111001101110000011011110111010 0

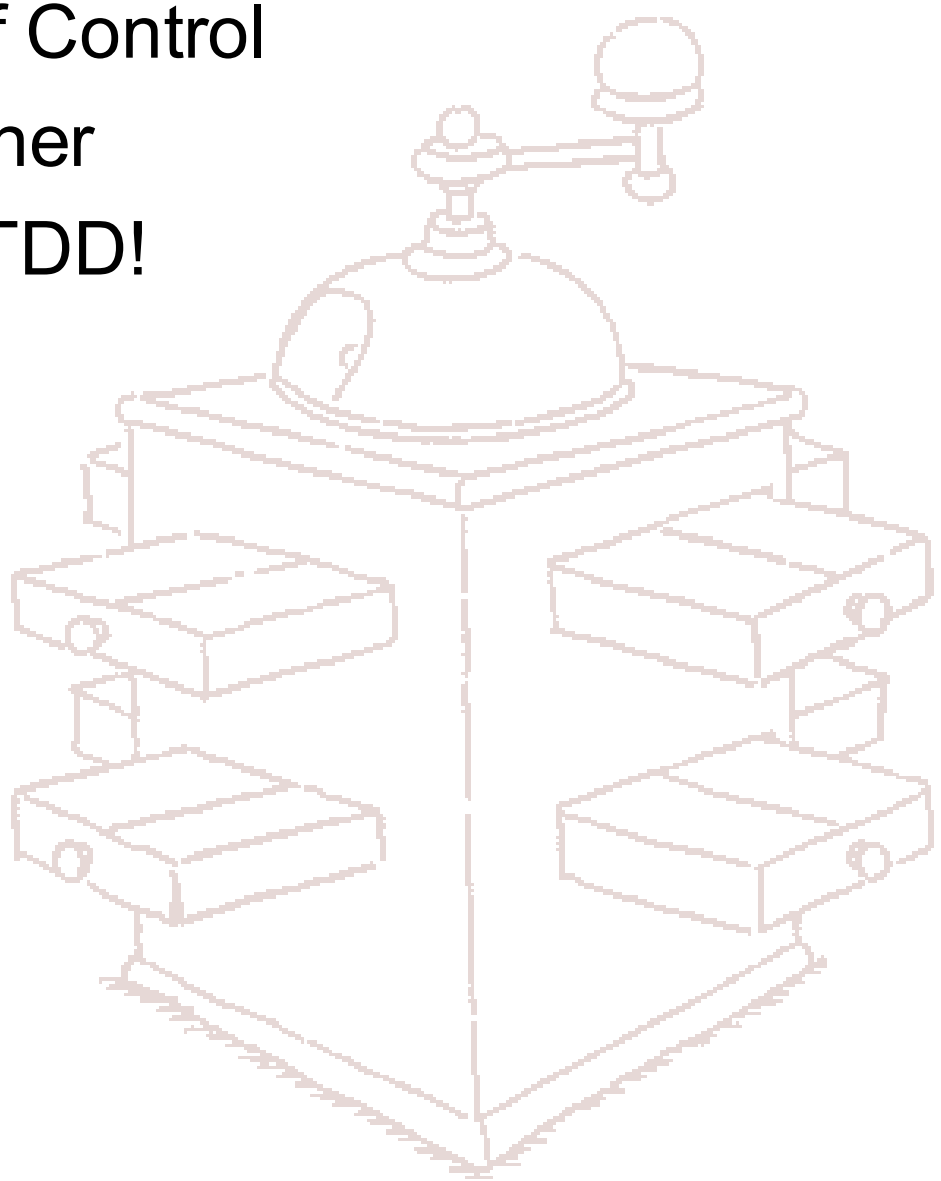
Paul couldn’t come…

“Inversion of Control is about software components doing what they are told, when they are told. Your OO application could well become unmaintainable without it.”

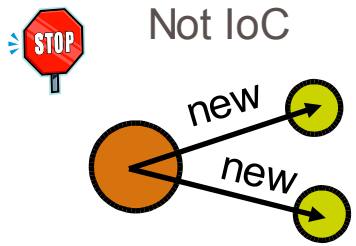
- Paul Hammant, ThoughtWorks

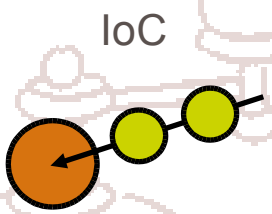
What we gonna talk about?

- Inversion of Control
- PicoContainer
- Code and TDD!
- Summary



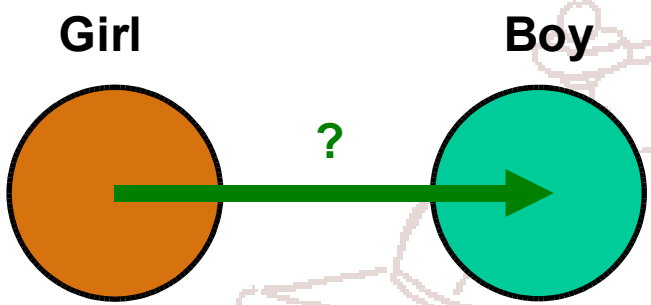
Inversion of Control





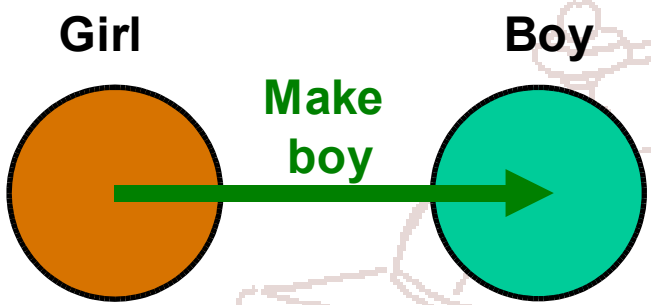
- **Testing** becomes easy
- **Maintenance** becomes easy
- **Configuration** becomes easy
- **Reuse** becomes easy

A simple system



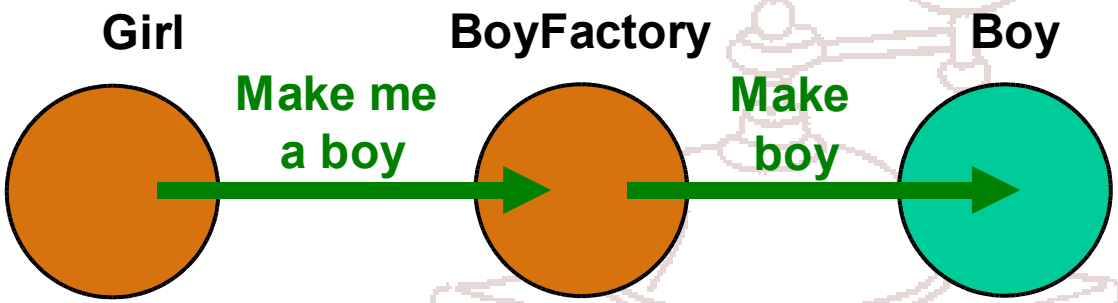
- A Girl that kisses a Boy
- How does the Girl get to know the Boy?

A simple system



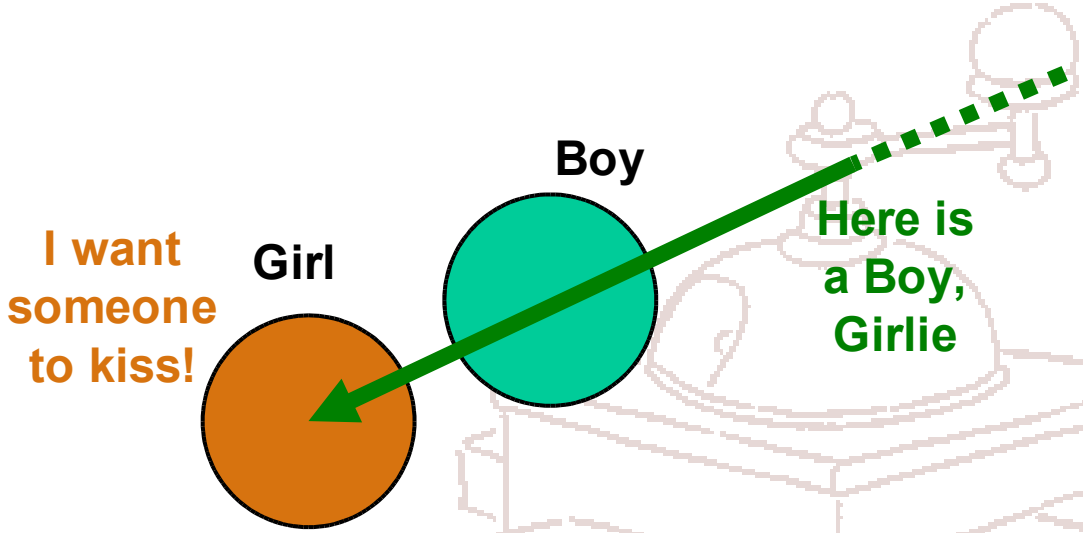
- Girl creates her own Boy…

Singleton or Factory



- Girl asks someone else to create a Boy
- Singleton is static and global
- So is the kind of Boy it creates
- Handier than “do it yourself”, but not flexible enough

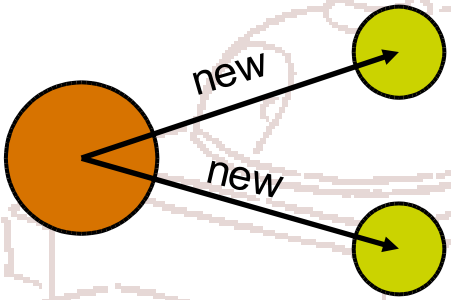
Inversion of Control



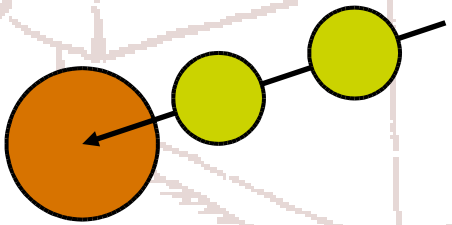
- It's no longer up to the Girl to find a Boy
- The Girl is *given* a Boy by someone else
- Also known as The Hollywood principle

The Hollywood Principle

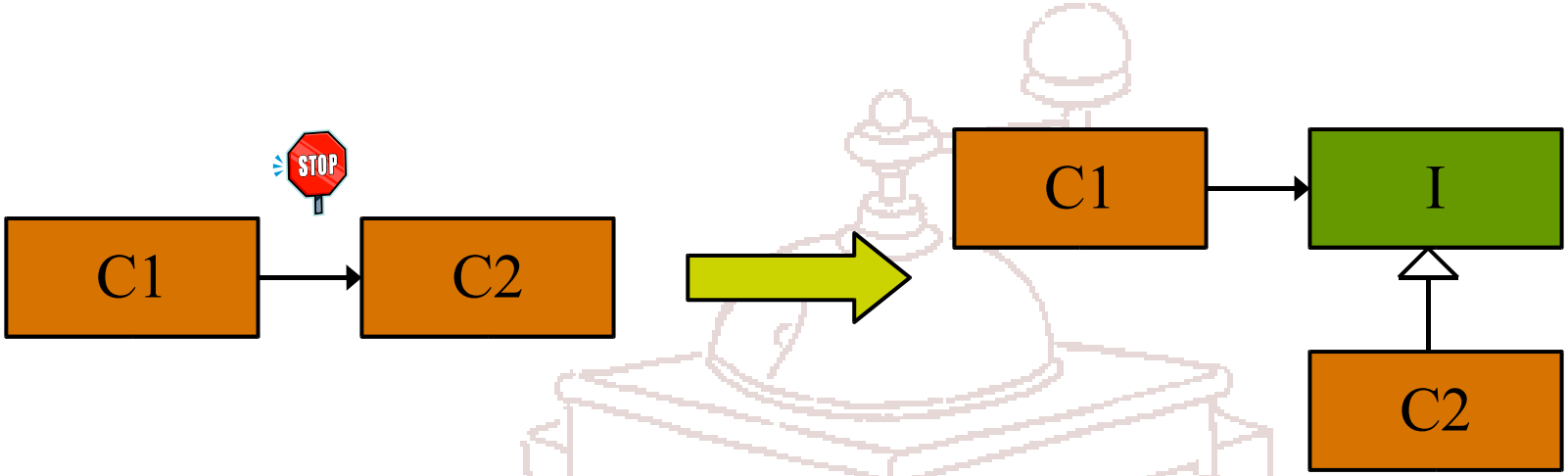
- Components do not reach out to the rest of the system to get dependencies



- Instead, they are handed their dependencies by an external entity



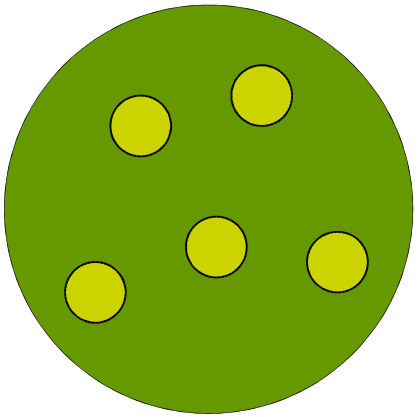
Dependency Inversion Principle



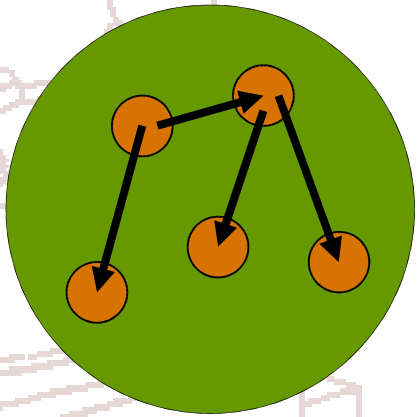
- Favours loose coupling
- Components should be split in two parts
  - **Service**, a declaration of offered functionality
  - **Implementation**, a specific implementation of a service
- Makes multiple runtime coupling combinations easy
- Breaks the dreaded “everything depends on everything” problem

PicoContainer

Containers



1. Register components

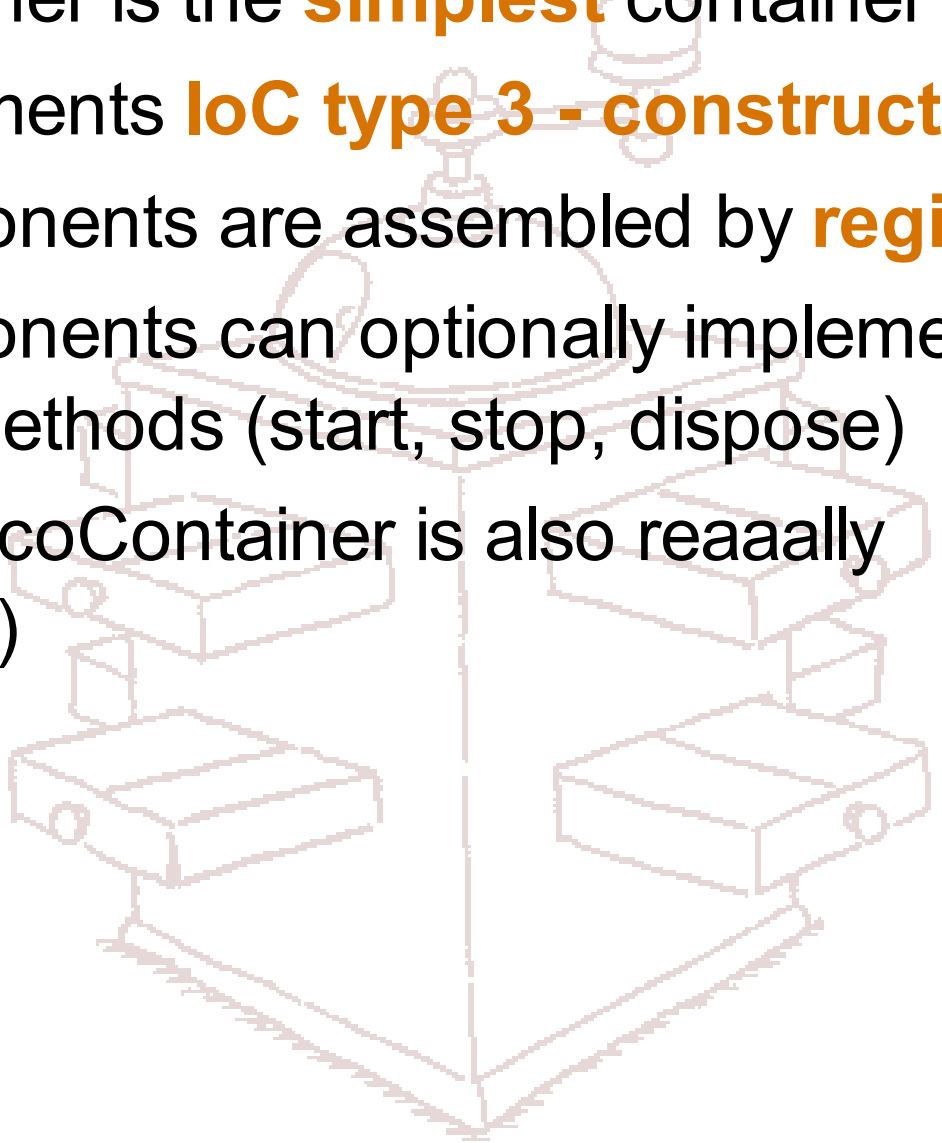


2. Materialize and lace the components

- Play the Hollywood role
- Reusable
- May provide other services
  - Lifecycle
  - Transactions
  - Etc…

So what about PicoContainer?

- PicoContainer is the **simplest** container for IoC
- Pico implements **IoC type 3 - constructors**
- Pico components are assembled by **registration**
- Pico components can optionally implement **lifecycle** methods (start, stop, dispose)
- (oh, btw, PicoContainer is also reaaally **extensible**)



Pico components are easy to write

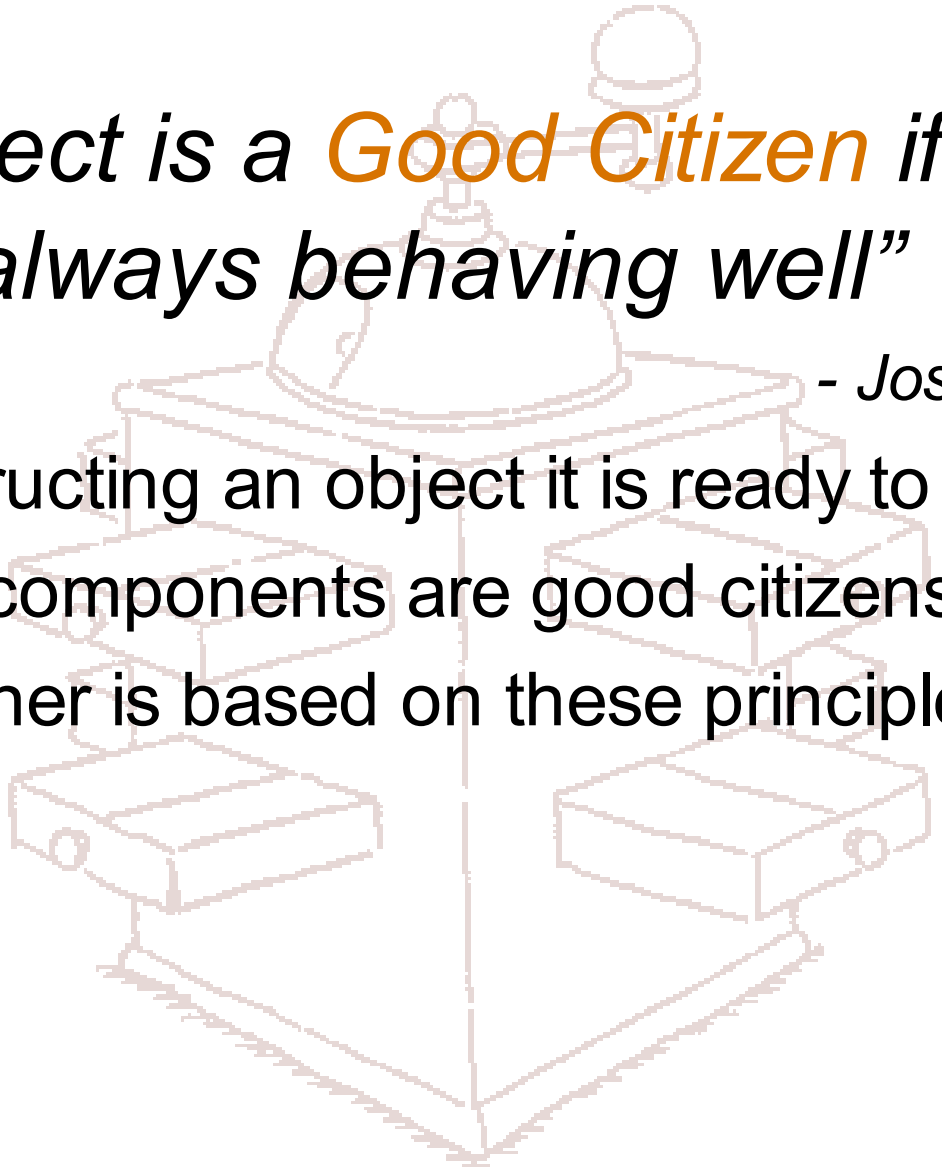
```
01 class Girl {
02     Kissable kissable;
03     Girl(Kissable kissable) {
04         this.kissable = kissable;
05     }
06     void kissYourKissable() {
07         kissable.kiss();
08     }
09 }
10
11 interface Kissable {
12     void kiss();
13 }
```

IoC type 3 is based on the Good Citizen Pattern

"An object is a Good Citizen if it is always behaving well"

- Joshua Bloch

- After constructing an object it is ready to go
- IoC type 3 components are good citizens
- PicoContainer is based on these principles



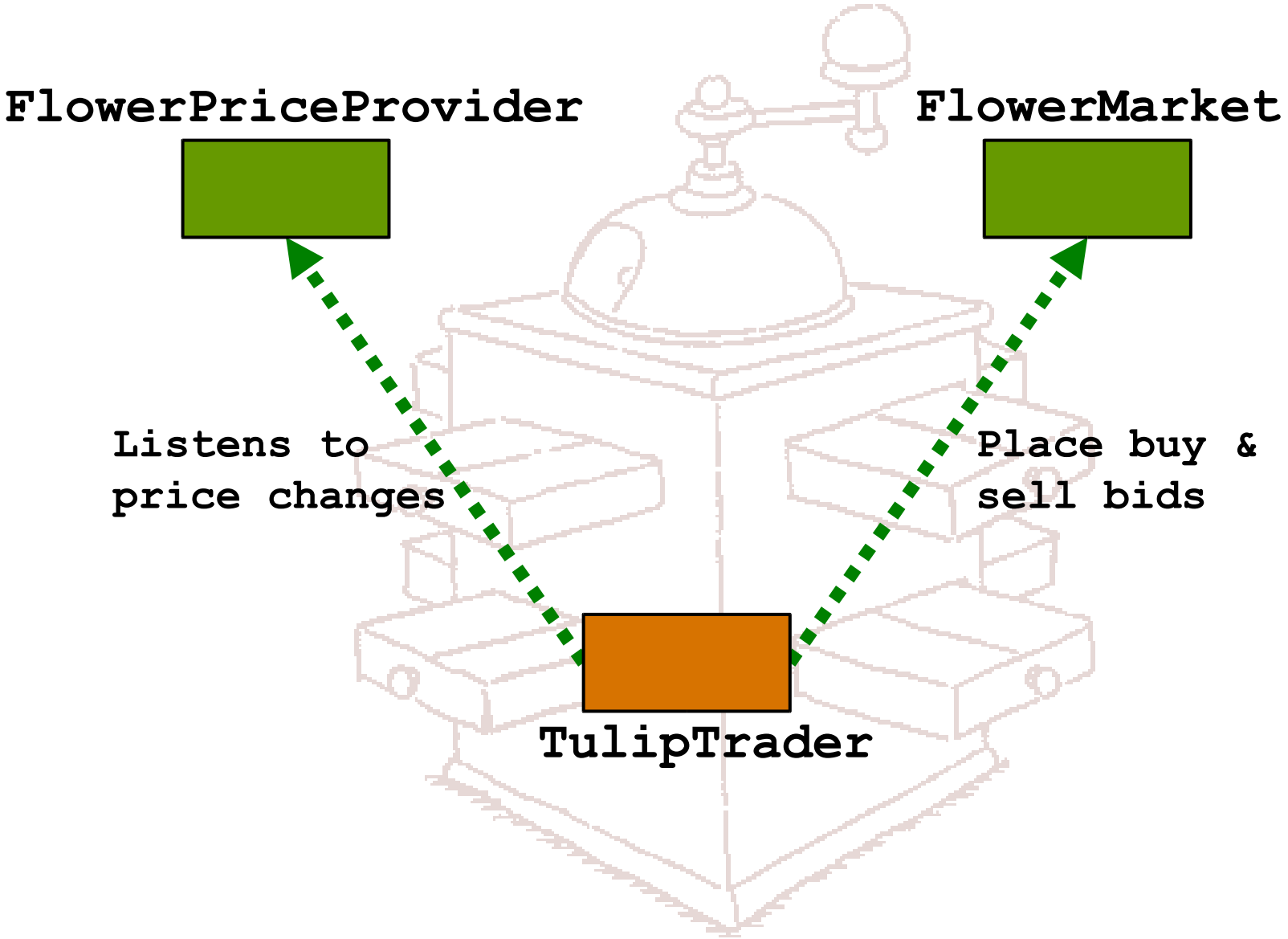
PicoContainer is simple to use

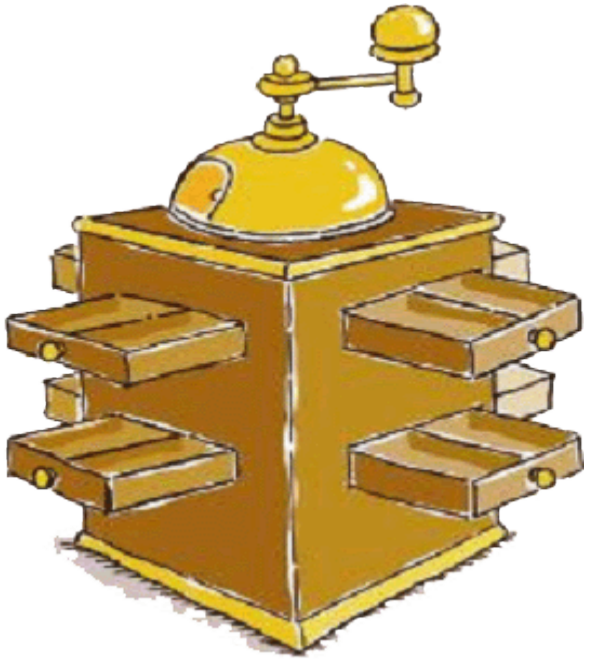
```
01 PicoContainer pico =
       new DefaultPicoContainer();
02 pico.regCI(Boy.class);
03 pico.regCI(Girl.class);
04 Girl girl = (Girl)
       pico.getCI(Girl.class);
05 girl.kissYourKissable();
```

```
regCI = registerComponentImplementation
getCI = getComponentInstance
```


BeJUG

Demo

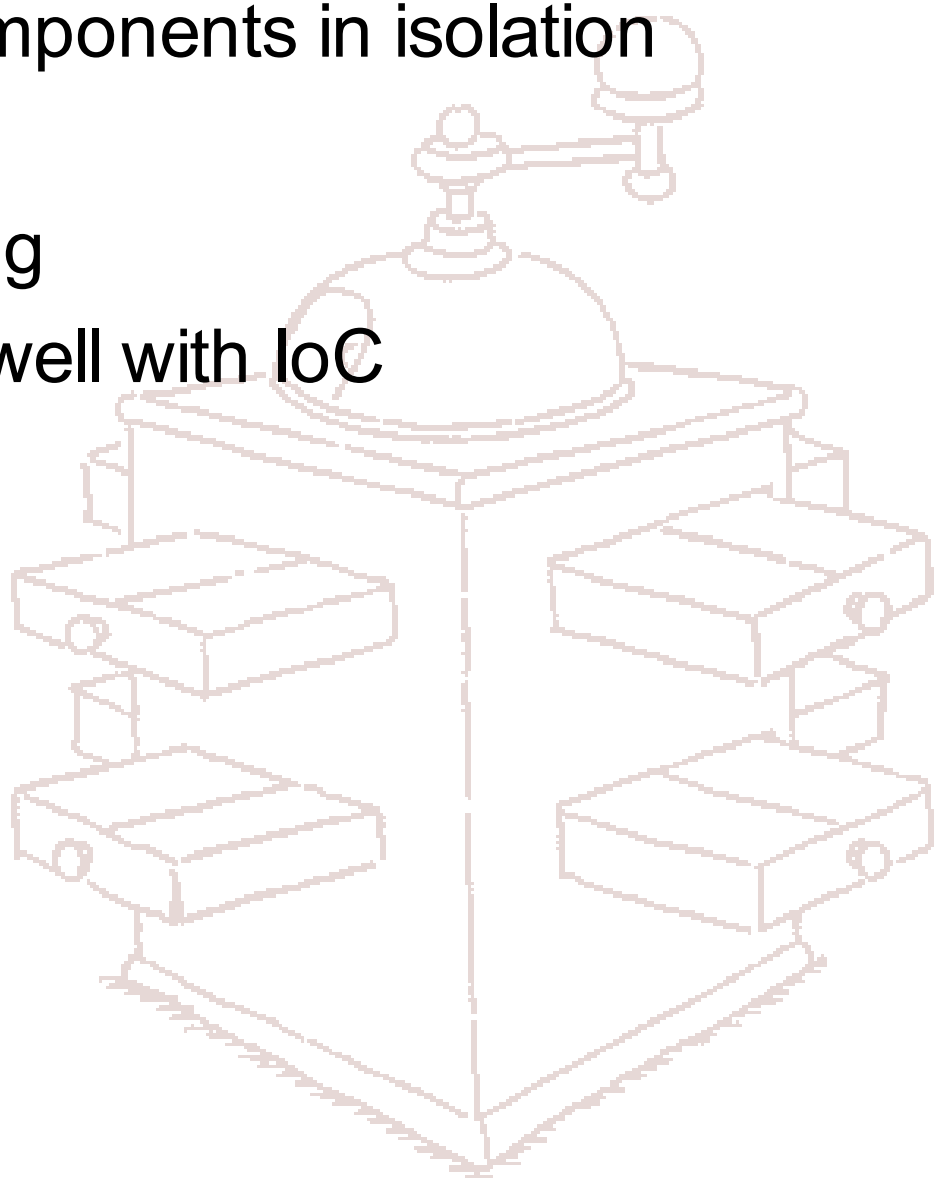

FlowerPriceProvider
FlowerMarket
Listens to price changes
Place buy & sell bids
TulipTrader



DEMO

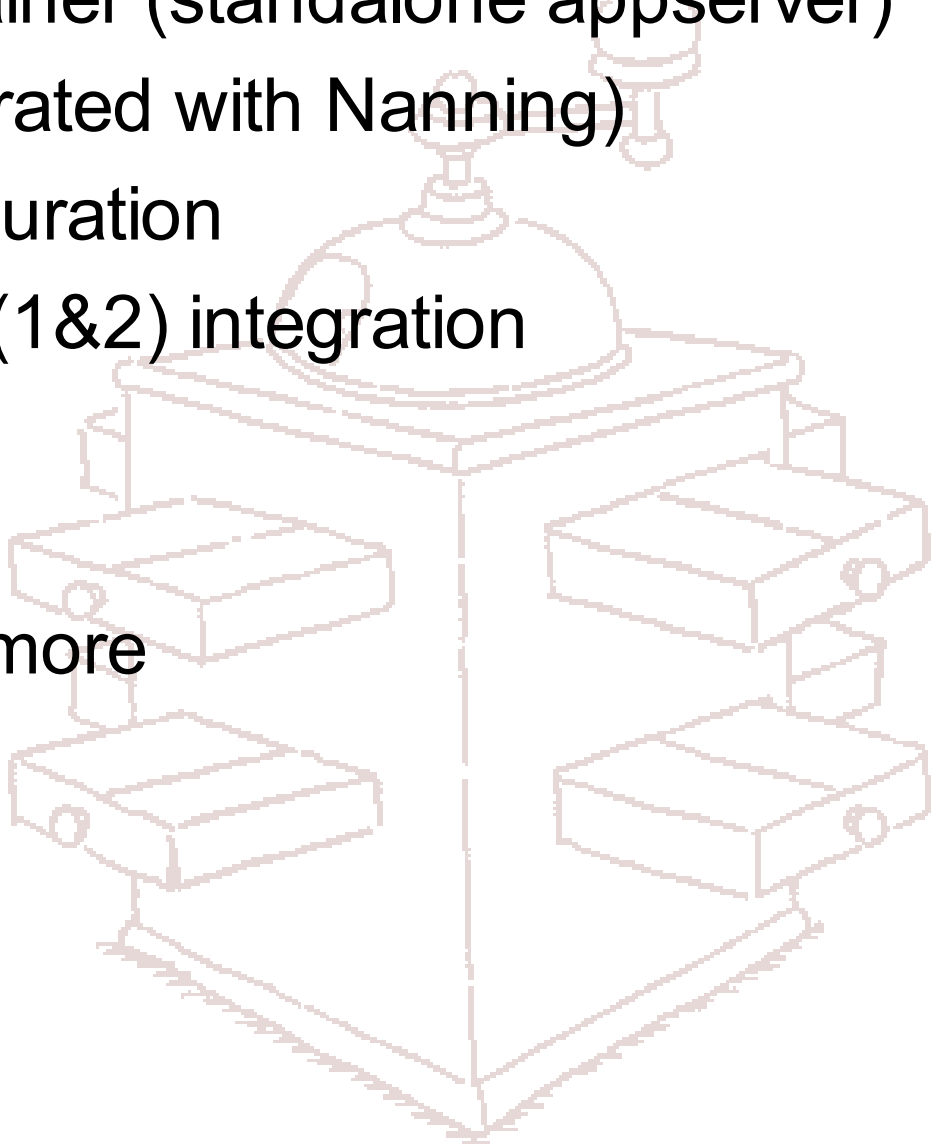
Mock objects

- Testing components in isolation
- Stub
- Endo-testing
- Goes very well with IoC

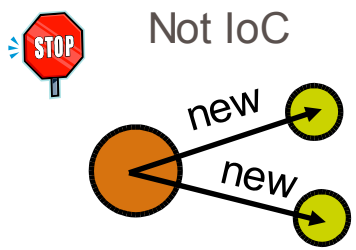


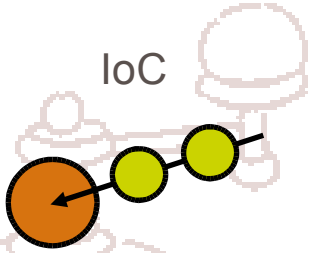
Extensions to PicoContainer

- NanoContainer (standalone appserver)
- AOP (integrated with Nanning)
- XML configuration
- WebWork (1&2) integration
- Ant task
- Pico GUI
- …and lots more

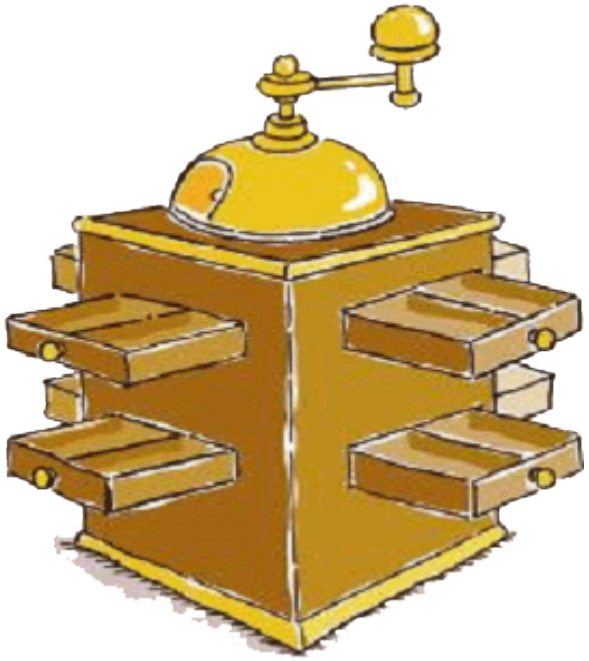


Summary – Inversion of Control





- **Testing** becomes easy
  - You can test the component in isolation by stubbing out entire parts of your application
- **Maintenance** becomes easy
  - Loose coupling facilitates local changes
- **Configuration** becomes easy
  - Component and service lacing is defined in one place
- **Reuse** becomes easy
  - A loosely coupled component can be reused outside its initial context



Q&A

Links

PicoContainer

http://picocontainer.org/

Codehaus

http://www.codehaus.org/

Avalon

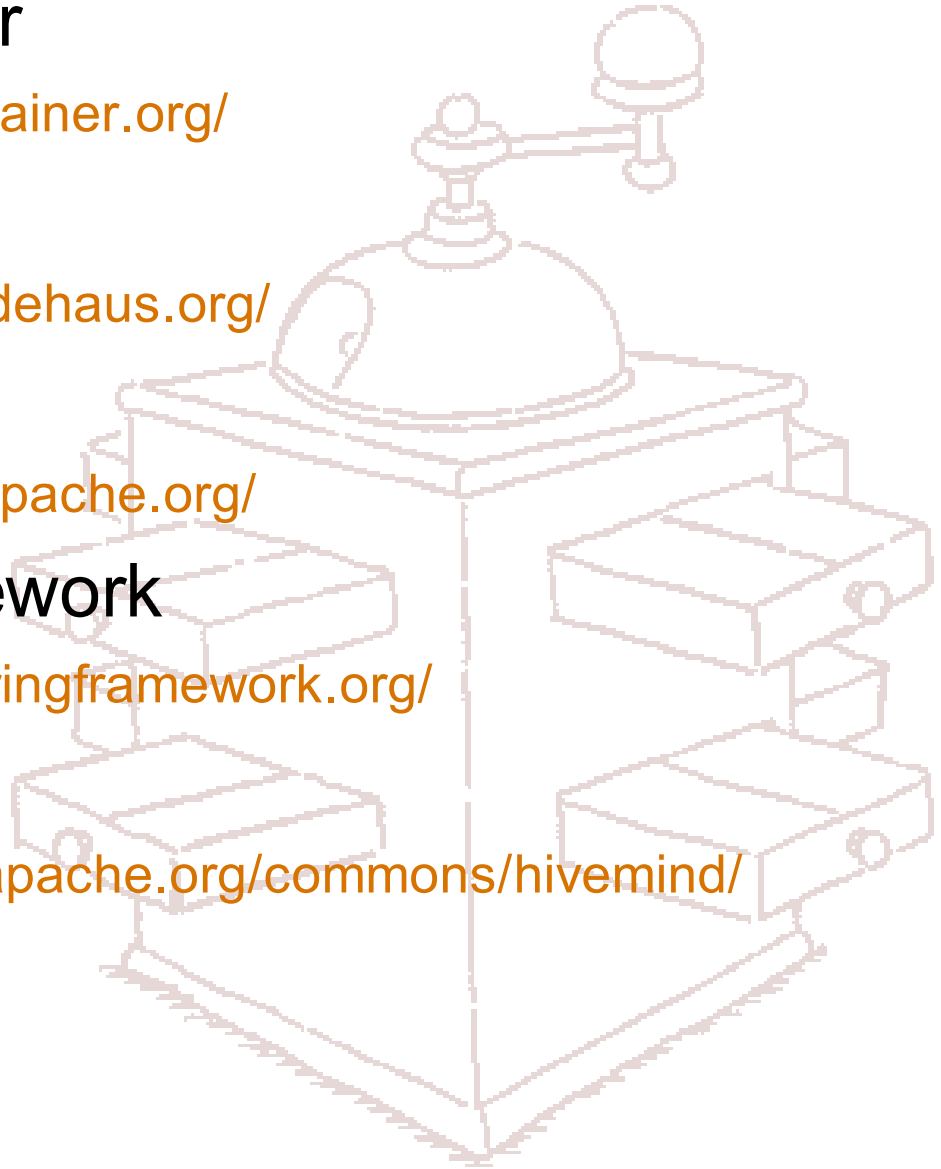
http://avalon.apache.org/

Spring Framework

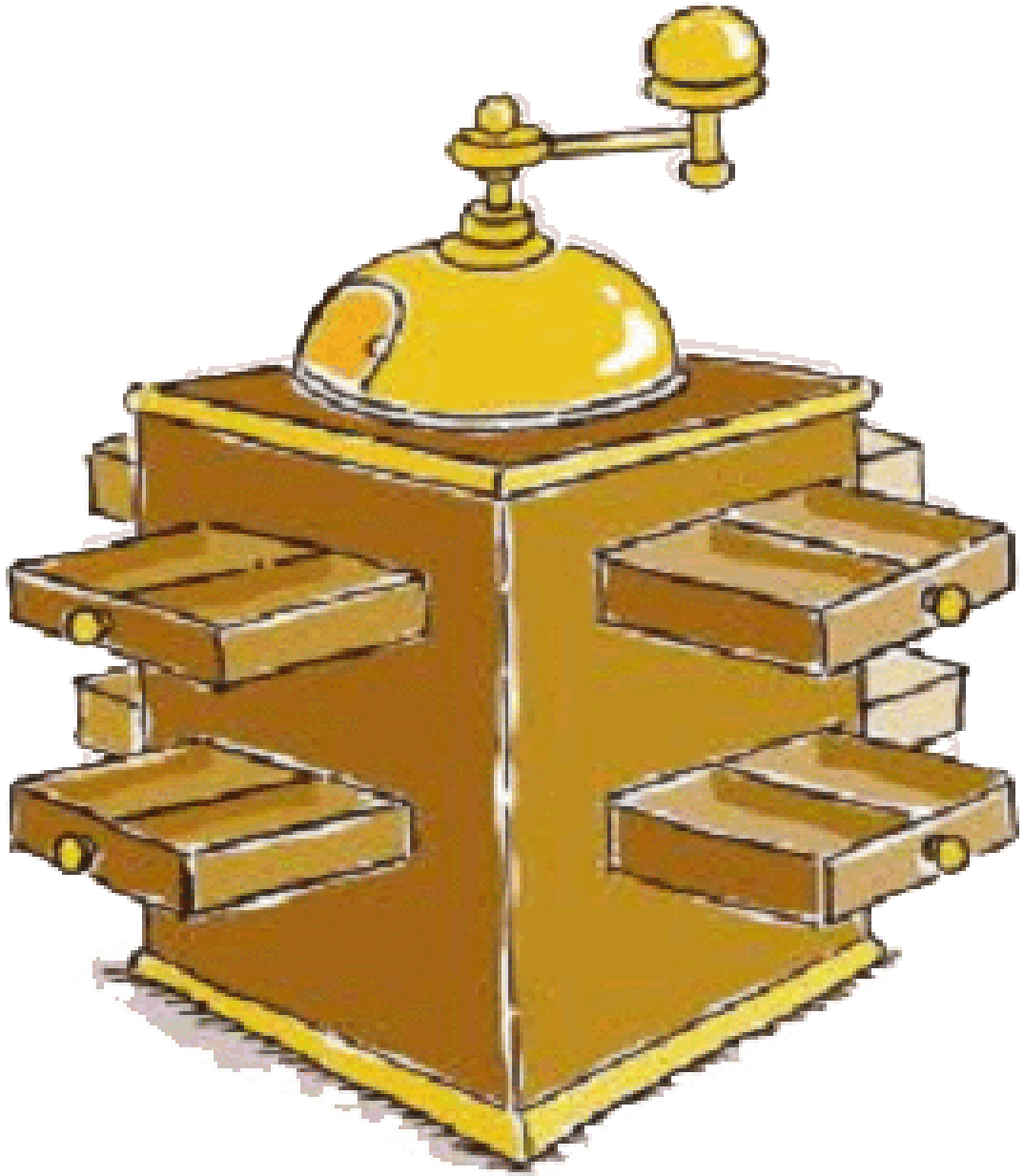
http://www.springframework.org/

HiveMind

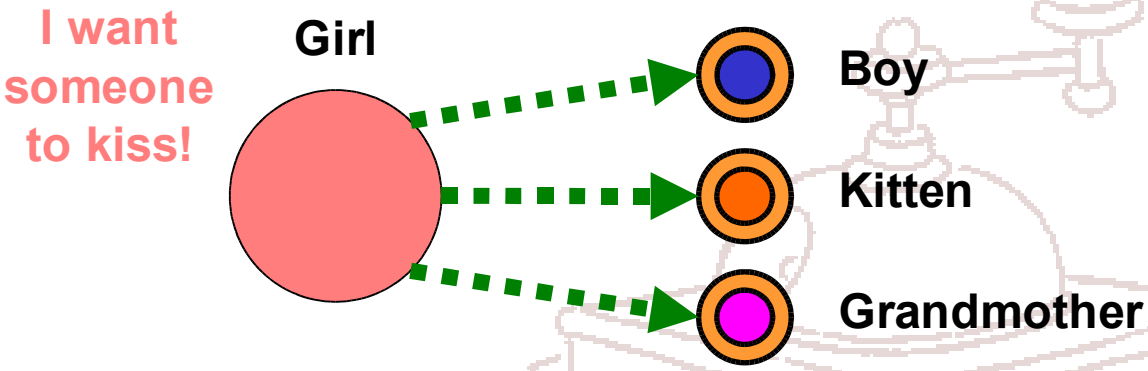
http://jakarta.apache.org/commons/hivemind/



JavaPolis 2003



So...



- If the Girls' primary concern is to have someone to *kiss*
  - she should declare that she needs a Kissable instead of a Boy
- A Girl can be fed a Boy, a Grandmother or a Kitten
- Flexibility!

IoC types

Type 1	Dependencies fetched from a ServiceManager or JNDI	Avalon, EJB/J2EE
Type 2	JavaBean setters with or without interfaces	Spring Framework, WebWork/XW
Type 3	Dependencies passed in constructor	PicoContainer, HiveMind

IoC type 0 – No IoC

No meta data, but you can't change the dependencies

```
public class Girl implements Servicable {
    Kissable kissable;
    public void service(ServiceManager mgr) {
        kissable = new Boy();
    }
    public void kissYourKissable() {
        kissable.kiss();
    }
}
```

No meta data, but you can't change the dependencies

IoC type 1 – Avalon example

Dependencies are fetched from a ServiceManager

```
public class Girl implements Servicable {
    Kissable kissable;
    public void service(ServiceManager mgr) {
        kissable = (Kissable) mgr.lookup("kissable");
    }
    public void kissYourKissable() {
        kissable.kiss();
    }
}
```

Hook up with meta-data

```
<container>
    <classloader> <classpath> … </classpath>
</classloader>
    <component name="kissable" class="Boy">
        <configuration> … </configuration>
    </component>
    <component name="girl" class="Girl" />
</container>
```

IoC type 2 – Spring example

Dependencies provided by JavaBean setters

```
public class Girl {
    Kissable kissable;
    public void setKissable(Kissable kissable) {
        this.kissable = kissable;
    }
    public void kissYourKissable() {
        kissable.kiss();
    }
}
```

Meta-data needed

```
<beans>
    <bean id="boy" class="Boy"/>
    <bean id="girl" class="Girl">
        <property name="kissable">
            <ref bean="boy"/>
        </property>
    </bean>
</beans>
```

IoC type 3 – PicoContainer example

Dependencies passed to the constructor

```
public class Girl {
    Kissable kissable;
    public Girl(Kissable kissable) {
        this.kissable = kissable;
    }
    public void kissYourKissable() {
        kissable.kiss();
    }
}
```

Meta-data or interfaces not needed (but supported)

```
PicoContainer container = new DefaultPicoContainer();
container.registerComponentImplementation(Boy.class);
container.registerComponentImplementation(Girl.class);
Girl girl = (Girl) container.getComponentInstance(Girl.class);
girl.kissYourKissable();
```