Elettra Sincrotrone Trieste

Elettra
Sincrotrone
Trieste

TANGO

School on TANGO Controls system

Design patterns

Giacomo Strangolino

IT programmer at Elettra – Sincrotrone Trieste

Assistant professor 2010-2014, University of Trieste,
Faculty of engineering, principles of computer science

mailto: giacomo.strangolino@elettra.eu
http://www.tango-controls.org

Design patterns

- Describe a *problem*;
- Describe a *solution*;

They help

- Find appropriate objects;
- Determine objects granularity and interface;
- Determine object dependencies;
- Make *object oriented* software **reusable** (*inheritance* vs. *composition*) and ***evolvable***;

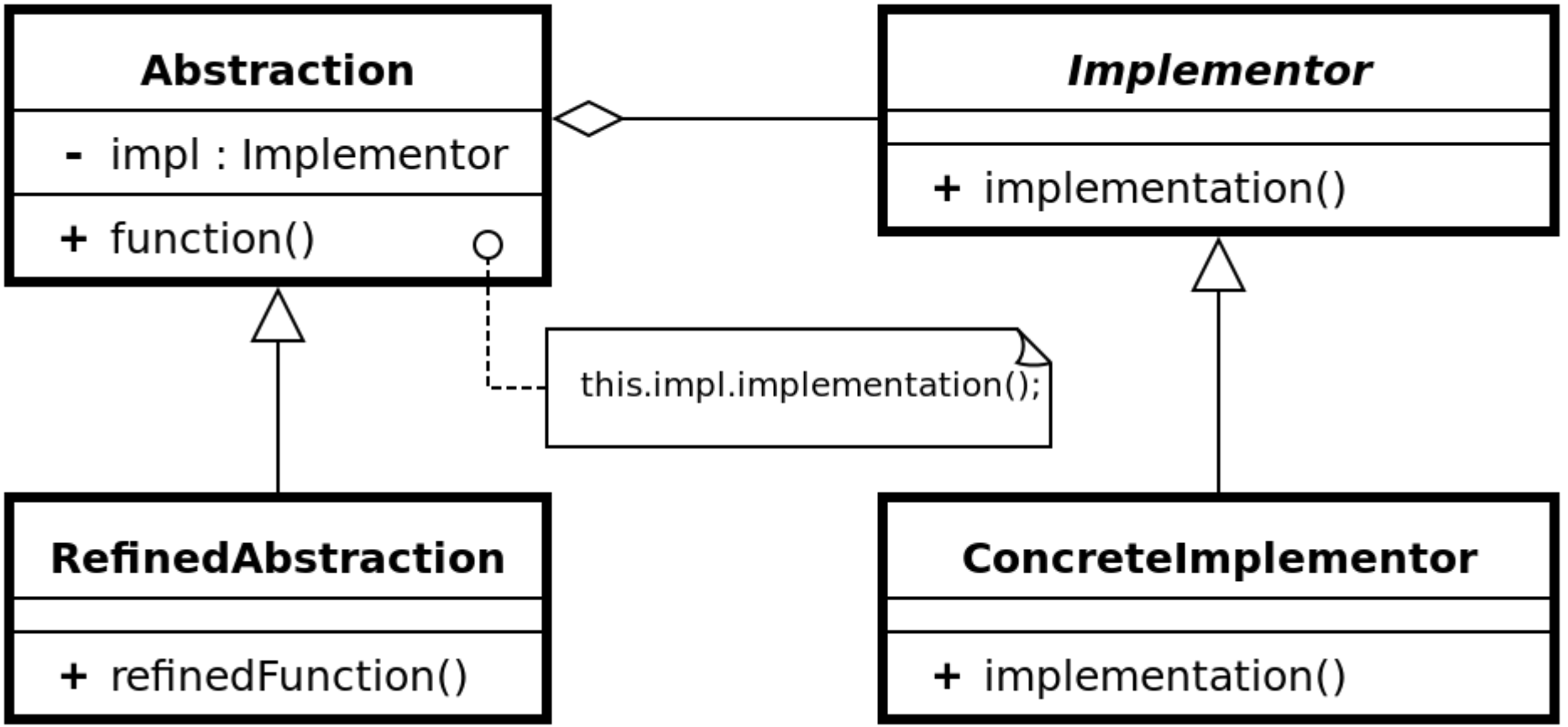
Design patterns

Example 1. Bridge

- Implementation switches at run time
- Abstractions *and* implementations can be extended by *subclassing*.
- Different abstractions and implementations can be combined;
- Changes in implementation do not affect clients *(binary compatibility!!!);*
- Hide implementation from clients

Design patterns

Example 1. Bridge



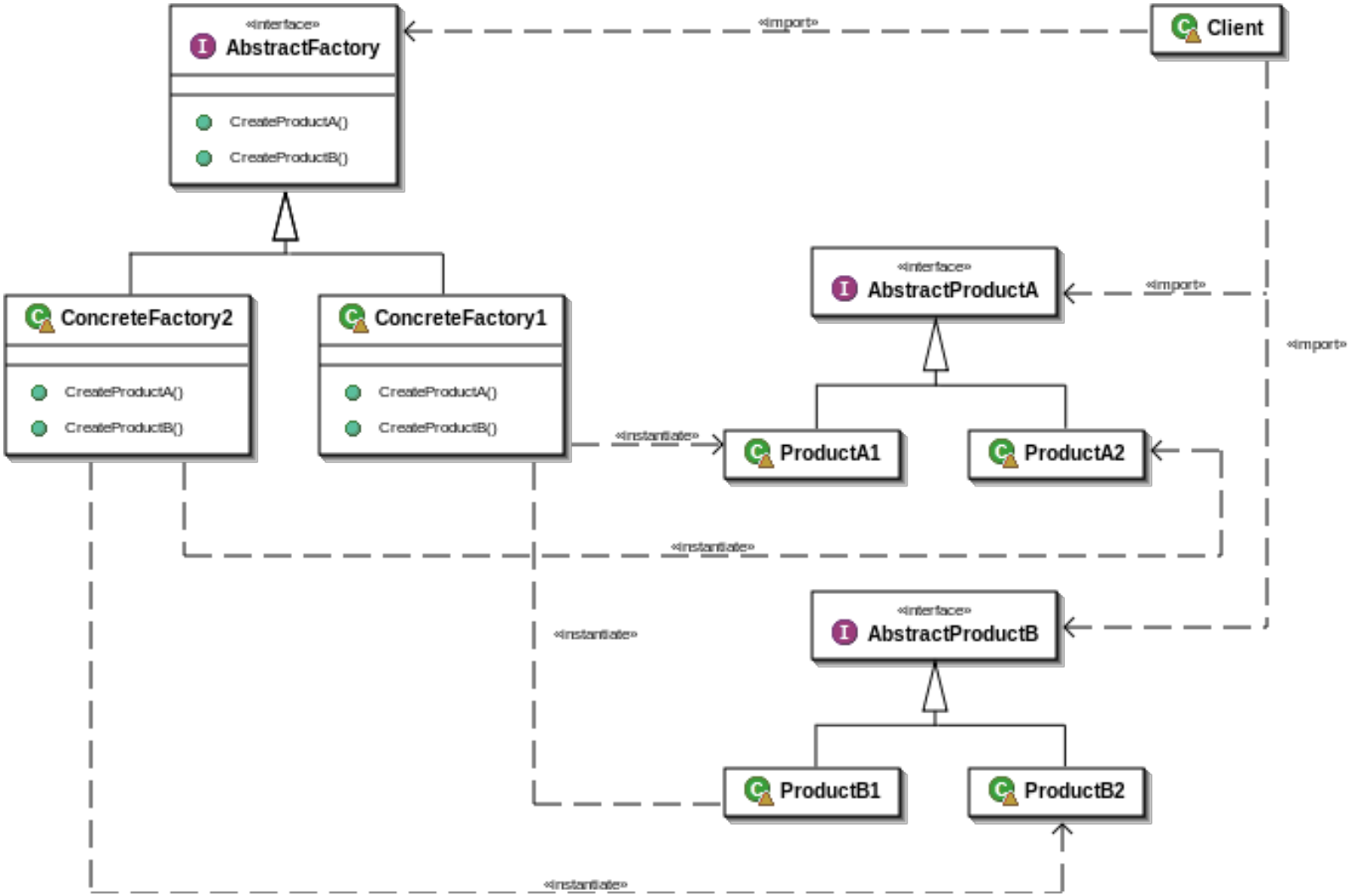
Design patterns

Example 2. Abstract factory

- Makes a system independent of how its products are created, composed, represented;
- A system can be configured with one of multiple families of products;
- A family of related products is designed to be used together;
- Provide a class library of products and reveal *just their interface*, not implementations.

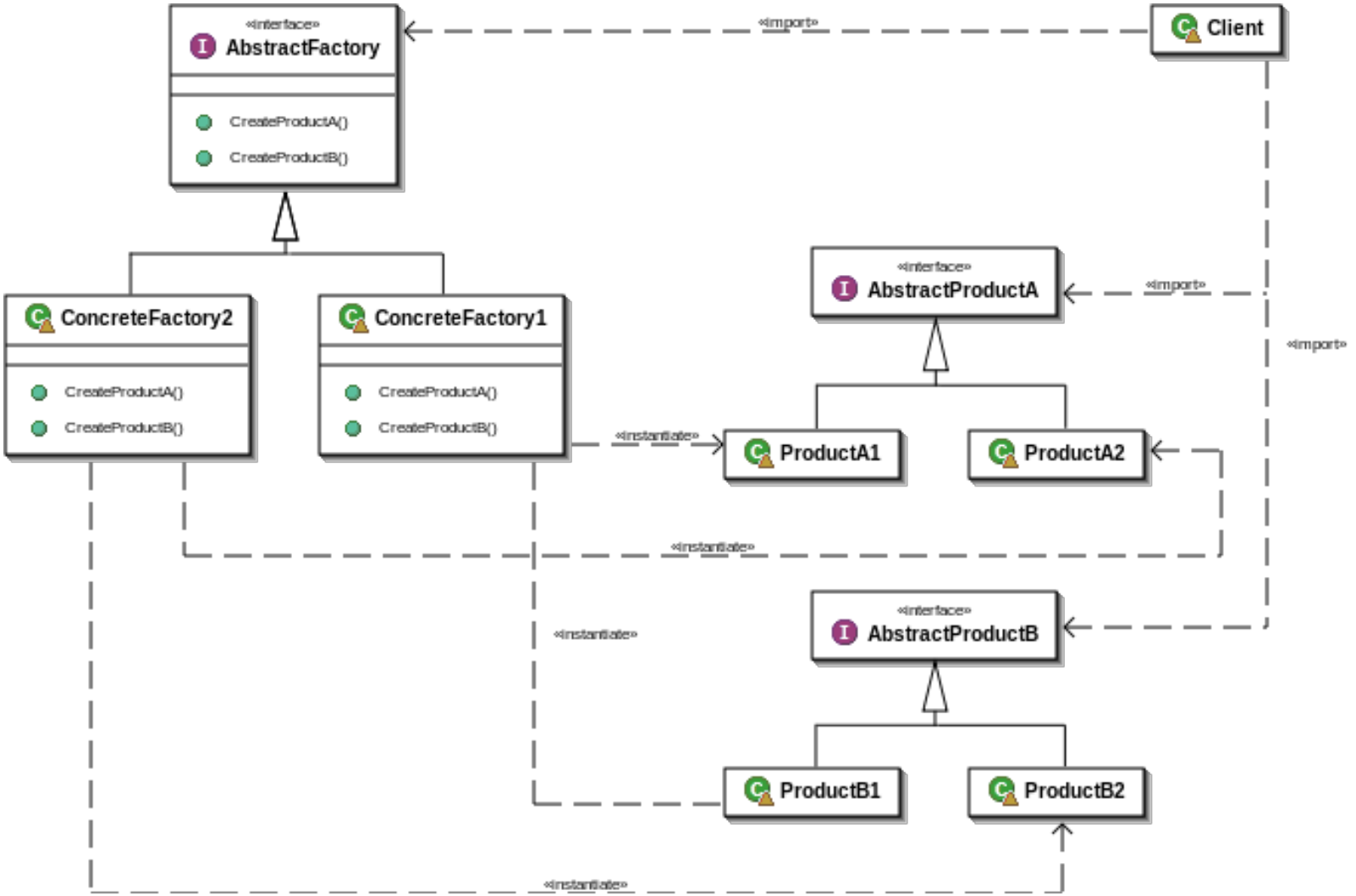
Design patterns

Example 2. Abstract factory



Design patterns

Example 2. Abstract factory



Design patterns

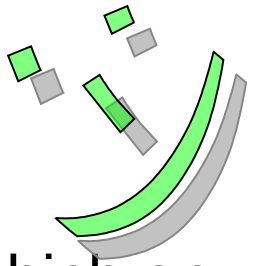
Example 2. Abstract factory

```
interface IButton
{
    void Paint();
}

interface IGUIFactory
{
    IButton CreateButton();
}

class WinFactory : IGUIFactory
{
    public IButton CreateButton()
    {
        return new WinButton();
    }
}

class OSXFactory : IGUIFactory
{
    public IButton CreateButton()
    {
        return new OSXButton();
    }
}
```

```
class WinButton : IButton
{
    public void Paint()
    {
        //Render a button in
    }
}

class OSXButton : IButton
{
    public void Paint()
    {
        //Render a button in
    }
}
```

```
class Program
{
    static void Main()
    {
        var appearance = Settings.Appearance;

        IGUIFactory factory;
        switch (appearance)
        {
            case Appearance.Win:
                factory = new WinFactory();
                break;
            case Appearance.OSX:
                factory = new OSXFactory();
                break;
            default:
                throw new System.NotImplementedException();
        }

        var button = factory.CreateButton();
        button.Paint();
    }
}
```

Design patterns



Example 3. Service locator

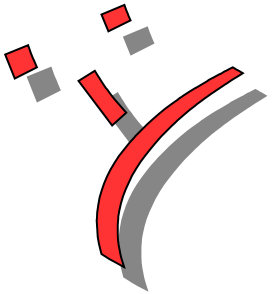
- Use a central *registry* known as ***service locator***, which on request returns the necessary objects to perform a task;
- It's a simple *run time linker*: code can be added at run time;
- Applications can select and remove items from the s. locator (replace a component with another one)
- Large sections of a library can be completely separated, the only link being the *service locator*.
- Model an object which is *singular in nature* (logging, memory management, audio device…)
- Can be applied to existing classes not designed around it (unlike *Singleton*).

Design patterns

Example 3. Service locator

- The registry must be unique (can be a bottleneck for concurrent applications)
- The registry hides the class' dependencies;
- The registry can be a security vulnerability: it allows outsiders to inject code into an application;
- Things placed in the registry are black boxes with regards to the rest of the system: harder to detect and recover from their errors

Use dependency injection!!



Design patterns

Example 3. Service locator

```
class Audio /* service interface */
{
    public:
      virtual void playSound(int soundID) = 0;
};

class ConsoleAudio : public Audio
{
public:
  virtual void playSound(int soundID)
  {
    // Play sound using console audio api...
  }
};
```

Design patterns

Example 3. Service locator

```
class Locator /* implementation of the service locator */
{
    public:
        static Audio getAudio() { return mService;  } /* does the locating */
        static void provide(Audio * service) { mService = service; }
    private:
        static Audio *mService;
};

class ConsoleAudio : public Audio
{
public:
  virtual void playSound(int soundID) { // Play sound using console audio api…}
};
```

Design patterns

Example 3. Service locator

- Register a provider before anything tries to use the service:

```
ConsoleAudio *audio = new ConsoleAudio();
Locator::provide(audio);
```

- Get the instance of audio service to use:

```
MyClass::MyClass() {
    Audio *audio = Locator::getAudio();
    audio->playSound(VERY_LOUD_BANG);
}
```

- The code calling *playSound()* is unaware of the concrete *ConsoleAudio* class.

Design patterns

Example 4. Dependency injection

```
Class MyClass{

    public:
        MyClass ( Audio *audio) { mAudio = audio;  }
    private:
        Audio *mAudio;
};

MyClass *myClass = new MyClass(new ConsoleAudio() );
```

A specific class instance (service) is *injected,* not created.

Dependency

- Control is inverted with respect to *Service locator*;
- Easy to test *MyClass*, providing a dummy *Audio* implementation.
- External code (*injector*) constructs the service and calls the client to inject it.

Design patterns

Bibliography

- E. Gamma, R. Helm, R. Johnson, J. Vlissides,

  Design Patterns – Elements of Reusable Object-Oriented software, Addison Wesley, 1998

- https://www.infoq.com/articles/Succeeding-Dependency-Injection

- http://gameprogrammingpatterns.com/service-locator.html

The End

- **Thanks for your attention**

mailto: giacomo.strangolino@elettra.trieste.it