Component Systems in the Field: Integrating and Controlling Operation Services easily using Connectors
Component Systems in the Field: Integrating and Controlling Operation Services easily using Connectors

Joachim Fröhlich, Florian Krautwurm, Markus Lachenmayr
Component systems in the field: Integrating and controlling operation services easily using connectors

By using connectors, a system improves its

- **Flexibility**
- **Structure**
- **Transparency**
Component systems in the field: Integrating and controlling operation services easily using connectors

Overview

- Application system characteristics
- Connectors – unloaded & loaded
- Upgrade of a system in operation
Component systems in the field: Integrating and controlling operation services easily using connectors

Overview

- Application system characteristics
- Connectors – unloaded & loaded
- Upgrade of a system in operation
**Application system characteristics**

What systems are we talking about here?

<table>
<thead>
<tr>
<th>Systems ...</th>
<th>Example: Mobile assembly robot¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>☑ ... control physical processes</td>
<td>☑ Collect and assemble workpieces in the production facility</td>
</tr>
<tr>
<td>☑ ... operate for long periods of time</td>
<td>☑ Navigate autonomously</td>
</tr>
<tr>
<td>☑ ... include small devices:</td>
<td>☑ Expensive battery charge</td>
</tr>
<tr>
<td>☑ little power</td>
<td></td>
</tr>
<tr>
<td>☑ small memory</td>
<td></td>
</tr>
<tr>
<td>☑ legacy code</td>
<td></td>
</tr>
<tr>
<td>☑ ... with expensive downtimes</td>
<td></td>
</tr>
</tbody>
</table>

¹ Amerongen J. (2006): The Role of Controls in Mechatronics, CRC Press
Adapt robots capabilities dynamically to new requirements

Update a robot’s navigation function for economic reasons

Assemble while driving

Situation
- Workpieces distributed in production facility
- Robot navigates autonomously
- Operation environment gets unstructured

Requirements
- Drive fast to the next workpiece
- Drive efficiently to the next workpiece
- Keep product quality high
## Application system requirements

Continuous system quality, also when system environment changes

<table>
<thead>
<tr>
<th>System services needed</th>
<th>Operation qualities needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitor resource utilization</td>
<td>Reliability</td>
</tr>
<tr>
<td>Profile and trace system usage</td>
<td>Availability</td>
</tr>
<tr>
<td>Test in the field, w/o side effects</td>
<td>Performance</td>
</tr>
<tr>
<td>Parallel deployments (blue/green)</td>
<td>Security</td>
</tr>
<tr>
<td>Canary releases</td>
<td>Resilience</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>
Component systems in the field: Integrating and controlling operation services easily using connectors

Overview

- Application system characteristics
- Connectors – unloaded & loaded
- Upgrade of a system in operation
Asymmetric interfaces couple components physically

Where do we come from?

Provider “owns” the interfaces

System build with components

1) Static linking for example and for clarity

$ cd Provider
$ g++ -c *.cpp -I. && ar r ../Provider.bin *.o
$ cd Client
$ g++ -c *.cpp -I../Provider && ar r ../Client.bin *.o
$ cd..
$ g++ -o System Client.bin Provider.bin
Symmetric interfaces in connectors
decouple components physically

Where are we going?

<table>
<thead>
<tr>
<th>Connector in between</th>
<th>System build with connectors(^1)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Diagram" /></td>
<td>$ cd Connector</td>
</tr>
<tr>
<td></td>
<td>$ g++ -c *.cpp -l. &amp;&amp; ar r ../Connector.bin *.o</td>
</tr>
<tr>
<td></td>
<td>$ cd Client</td>
</tr>
<tr>
<td></td>
<td>$ g++ -c *.cpp -l./Connector &amp;&amp; ar r ../Client.bin *.o</td>
</tr>
<tr>
<td></td>
<td>$ cd Provider</td>
</tr>
<tr>
<td></td>
<td>$ g++ -c *.cpp -l./Connector &amp;&amp; ar r ../Provider.bin *.o</td>
</tr>
<tr>
<td></td>
<td>$ cd ..</td>
</tr>
<tr>
<td></td>
<td>$ g++ -o Connector.bin Provider.bin System Client.bin</td>
</tr>
</tbody>
</table>

1) Static linking for example and for clarity
Applications insulated from implementations

Client view on a connector

### Connector hides provider(s)

- **Client**
  - `ref`
  - `navigate`

- **RS17**
  - `Steering`

- **Connector**
  - `navigate`

- **Chassis**
  - `navigate`

### Use interface

1: `Navigate* n = Chassis::my() † navigate();`
2: `Route* r = n † navigate(w);`
   
   // Plan route to the next workpiece

---

---

---

---

1) RoboSat17  2) Client-specific interface  3) C++: #define Interface struct  4) C++: #define Class class

---

Unrestricted © Siemens AG 2017

Markus Lachenmayr | Siemens Corporate Technology

Page 12 May 2, 2017
## Insulating connector enables fully transparent services

View into an unloaded connector

### Factory bisected to ...

<table>
<thead>
<tr>
<th>Chassis</th>
<th>navigate</th>
<th>Navigate</th>
</tr>
</thead>
<tbody>
<tr>
<td>bisect</td>
<td>implement</td>
<td>Navigate Default</td>
</tr>
</tbody>
</table>

### ... create defaults or test doubles

```cpp
1: Class Chassis {
2:   public: Navigate* navigate();\(^1\) // Factory
3:   protected: virtual Navigate* implement();\(^2\)
     // Factory, provider part
4:   };

5: Navigate* Chassis::navigate() {
6:   return this\(\ast\) implement();
7: }
8: Navigate* Chassis::implement() {
9:   return new NavigateDefault();
     // Standalone factory, independent connector
10: }
```
Insulating connector enables fully transparent services

View into a loaded connector

**Factory bisected ...**

![Diagram of a factory bisected with navigate and inject actions]

**... to inject servicing proxies**

```cpp
1: Class Chassis {
2:     public: Navigate* navigate();\(^1\)
3:     protected: virtual Navigate* implement();\(^2\)
4: };
5: Navigate* Chassis::navigate() {
6:     Navigate* n = this->implement();
7:     return new SwitchNavigation(n); // Proxy
8: }

1: Class RS17Chassis : Chassis { ... };
2: Navigate* RS17Chassis::implement() {
3:     return new FastRoute(); // Override
4: }
```

\(^1\) Referenced in .h
\(^2\) Referenced in .C
### Implementing connectors with well-proven and established patterns

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Intention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separated Interface(^{Fowler})</td>
<td>Define interfaces in separate binaries.</td>
</tr>
<tr>
<td>Insulation(^{Lakos})</td>
<td>Decouple clients and providers. Optionally, integrate services without recompiling clients and providers.</td>
</tr>
<tr>
<td>Abstract Factory(^{GoF})</td>
<td>Defer object creation to (1) decouple clients and providers and to (2) optionally insert proxies.</td>
</tr>
<tr>
<td>Registry(^{Fowler})</td>
<td>Providers register and clients find concrete factories dynamically.</td>
</tr>
<tr>
<td>Proxy(^{GoF})</td>
<td>Integrate operation service around interfaces.</td>
</tr>
<tr>
<td>Chain of Responsibility(^{GoF})</td>
<td>Integrate several operation services.</td>
</tr>
<tr>
<td>Strategy(^{GoF})</td>
<td>Control clients, providers, services in connectors.</td>
</tr>
<tr>
<td>Gateway(^{Fowler})</td>
<td>Control connectors remotely.</td>
</tr>
</tbody>
</table>
### Operation services in connectors

Loaded connectors can …

<table>
<thead>
<tr>
<th>... handle several clients</th>
<th>... provide different services</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drive fast</td>
<td>Monitor</td>
</tr>
<tr>
<td>Drive efficiently</td>
<td>wel Switch</td>
</tr>
<tr>
<td>Navigation : Connector</td>
<td>Navigation : Connector</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>... handle several providers</th>
<th>... be controlled dynamically</th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigation : Connector</td>
<td>Switch (Fast or Eco)</td>
</tr>
<tr>
<td>FastRoute</td>
<td>Test (Fast or Eco)</td>
</tr>
<tr>
<td>EfficientRoute</td>
<td>Navigation : Connector</td>
</tr>
</tbody>
</table>
Connectors link Development and Operations

Loaded connectors can be controlled dynamically and remotely
Component systems in the field: Integrating and controlling operation services easily using connectors

Overview

- Application system characteristics
- Connectors – unloaded & loaded
- Upgrade of a system in operation
Upgrade of a mobile assembly robot in operation

The requirements that shape the Simplex architecture\(^1\) come close to ours

<table>
<thead>
<tr>
<th>Robot requirements</th>
<th>Simplex requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>⃣ Test two routing functions in the field</td>
<td>⃣ Change function units online safely</td>
</tr>
<tr>
<td>⃣ Switch dynamically</td>
<td>⃣ Provide units that can be modified and replaced online with facilities to make the change simple</td>
</tr>
</tbody>
</table>

Orthogonal operation services in connectors

Transparent Blue/Green deployments with connectors

**Run blue and test green**

- Control Strategy
  - Switch = Fast
  - Test = Eco

- Switch

- Test

- FastRoute

- EcoRoute

**Run green, option to switch to blue**

- Control Strategy
  - Switch = Eco
  - Test =

- Switch

- Test

- FastRoute

- EcoRoute
Orthogonal operation services in connectors

Services for durable control systems of high quality and quick on the market

Operation services as needed
- Blue/green deployment
- A/B test
- Canary release
- Monitor, profile and trace
- Budget
- Parallel execution
- Test regressions
- …
Connectors simplify Development and Operations

Symmetric interfaces in connectors that decouple components physically

<table>
<thead>
<tr>
<th>Central connectors</th>
<th>Easier and faster development</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Diagram" /></td>
<td>✅ Focus on domain concepts &amp; relations</td>
</tr>
<tr>
<td></td>
<td>✅ Modular development and test</td>
</tr>
<tr>
<td></td>
<td>✅ Technologies and details excluded</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Easier and maintainable operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>✅ Services separated from functions</td>
</tr>
<tr>
<td>✅ Modular deployment in the field</td>
</tr>
<tr>
<td>✅ Faster, exact feedback from the field</td>
</tr>
</tbody>
</table>
Connectors link Development and Operations

Build, integration and test processes reach the operation environment

<table>
<thead>
<tr>
<th>Development</th>
<th>Infrastructure</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>€ Domain-driven</td>
<td>€ Connectors</td>
<td>€ High availability</td>
</tr>
<tr>
<td>€ Modular and insulated</td>
<td>€ Connector Manager</td>
<td>€ Faster feedback</td>
</tr>
<tr>
<td>€ Testable</td>
<td>€ Gateway to CI Server</td>
<td>€ Exact field data</td>
</tr>
</tbody>
</table>

- Connectors
- Connector Manager
- Gateway to CI Server

Diagram:
- Story 201
- Story 197
- Spike 23
- Story 142
- CI Server
- Provider A1
- Provider A2
- Provider B1
- Connector A
- Connector B
- Gateway
- Client
Component systems in the field: Integrating and controlling operation services easily using connectors

Characteristics of connectors in this presentation

- **Flexibility**: Organize interfaces in binaries, independently from function components.
- **Structure**: Add services around interfaces, orthogonally to component functions.
- **Transparency**: Monitor, test, secure, load, change, budget … component functions transparently, during system operation.
Component Systems in the Field:
Integrating and Controlling System Services easily using Connectors

Joachim Fröhlich
Florian Krautwurm
Markus Lachenmayr

Otto-Hahn-Ring 6
81739 Munich

+49 (89) 636 630741
markus.lachenmayr@siemens.com
siemens.com/corporate-technology
Component Systems in the Field: Integrating and Controlling Operation Services easily using Connectors
Appendix: References

Gamma E., et al. (1995): Design Patterns. Addison-Wesley (GoF)
Hohpe G., Woolf B. (2004): Enterprise Integration Patterns. Addison-Wesley
Lakos J. (1996): Large-Scale C++ Software Design. Addison-Wesley
Component systems in the field: Integrating and controlling operation services easily using connectors

Overview

- Application system characteristics
- Connectors – unloaded & loaded
- Upgrade of a system in operation
- Component and connector views
## Component and connector views

Connectors in this presentation in relation to …

<table>
<thead>
<tr>
<th>... component and connector views</th>
<th>... patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>⌂ Connectors are boxes, not lines</td>
<td>⌂ Interceptors are for networking objects</td>
</tr>
<tr>
<td>⌂ Connectors drive architectural styles</td>
<td>⌂ Smart proxies monitor networks</td>
</tr>
<tr>
<td>⌂ No connectors in the 4C model</td>
<td>⌂ Proxies in connectors like smart pointers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>... middleware</th>
<th>... techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>⌂ Overly complex technologies for embedded systems</td>
<td>⌂ Interface-based programming: Programming with connectors!</td>
</tr>
<tr>
<td>⌂ Connectors can be integrated</td>
<td>⌂ AOP: weave services into connectors</td>
</tr>
</tbody>
</table>

2) 4C = Context, Container, Components, Classes from Brown S. (2017): Software Architecture as Code. OOP Munich
Implementations insulated from applications

Provider view on a connector

**Connector hides client(s)**

**Implement interface**

1: Class Chassis {
2:   protected: virtual Navigate* implement();\(^1\)
\[1\] Factory, provider interface
3: };
4: Class RS17Chassis: Chassis { ... };
5:   Navigate* RS17Chassis::implement() {
6:     return new FastRoute();
7:   }
8: Class FastRoute: Navigate {
9:   Route* navigate(Workpiece*);
0: }

---

1) Provider-specific interface 2) RoboSat17

Unrestricted © Siemens AG 2017

Page 30 May 2, 2017

Markus Lachenmayr | Siemens Corporate Technology
Insulating connector enables fully transparent services

View into an unloaded connector

### Separated interfaces

<table>
<thead>
<tr>
<th>Chassis</th>
<th>navigate</th>
<th>Navigate</th>
</tr>
</thead>
<tbody>
<tr>
<td>bisect</td>
<td>implement</td>
<td>0</td>
</tr>
</tbody>
</table>

### Standalone factories

```cpp
1: Class Chassis {
2:   public: Navigate* navigate();  // Factory
3:   protected: virtual Navigate* implement();  // Factory, provider interface
4:     }
5:   Navigate* Chassis::navigate() {
6:      return this->implement();
7:   }
8:   Navigate* Chassis::implement() {
9:      return 0;  // Makes the factory standalone and the connector provider-independen
0:   }
```
Insulating connector enables fully transparent services

View into a connector

**Hidden and lazy factory creation ...**

```
1: Class Chassis {
2:   public: static Chassis* my();  // Hide factory
3:   protected: Chassis();          // Auto register
4:   };
5: static Chassis* c = 0;
6: Chassis::Chassis() { c = this; } // Auto register provider
7: Chassis::my() {
8:   if (0 == c) { c = new Chassis(); } // Default
9:   return c;
10: }
11: Class RS17Chassis : Chassis { … };
12: static RS17Chassis r;  // Auto register provider
```

... to enable auto-registration

```
register default
if no provider
has registered yet
```

```
register
```
Implementing connectors with well-proven and established patterns

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Intention</th>
<th>U</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separated Interface(^{Fowler})</td>
<td>Define interfaces in separate binaries.</td>
<td>r</td>
<td>r</td>
</tr>
<tr>
<td>Insulation(^{Lakos})</td>
<td>Decouple clients and providers. Optionally, integrate services without recompiling clients and providers.</td>
<td>r</td>
<td>r</td>
</tr>
<tr>
<td>Abstract Factory(^{GoF})</td>
<td>Defer object creation to (1) decouple clients and providers and to (2) optionally insert proxies.</td>
<td>r</td>
<td>r</td>
</tr>
<tr>
<td>Registry(^{Fowler})</td>
<td>Providers register and clients find concrete factories dynamically.</td>
<td>r</td>
<td>r</td>
</tr>
<tr>
<td>Proxy(^{GoF})</td>
<td>Integrate operation service around interfaces.</td>
<td>na</td>
<td>r</td>
</tr>
<tr>
<td>Chain of Responsibility(^{GoF})</td>
<td>Integrate several operation services.</td>
<td>na</td>
<td>o</td>
</tr>
<tr>
<td>Strategy(^{GoF})</td>
<td>Control clients, providers, services in connectors.</td>
<td>na</td>
<td>o</td>
</tr>
<tr>
<td>Gateway(^{Fowler})</td>
<td>Control connectors remotely.</td>
<td>na</td>
<td>o</td>
</tr>
</tbody>
</table>

U = Unloaded connectors, L = Loaded connectors, r = required, o = optional, na = not available