

**Effective Component and Application
Development using the Software
Communication Architecture
(Part 1)**

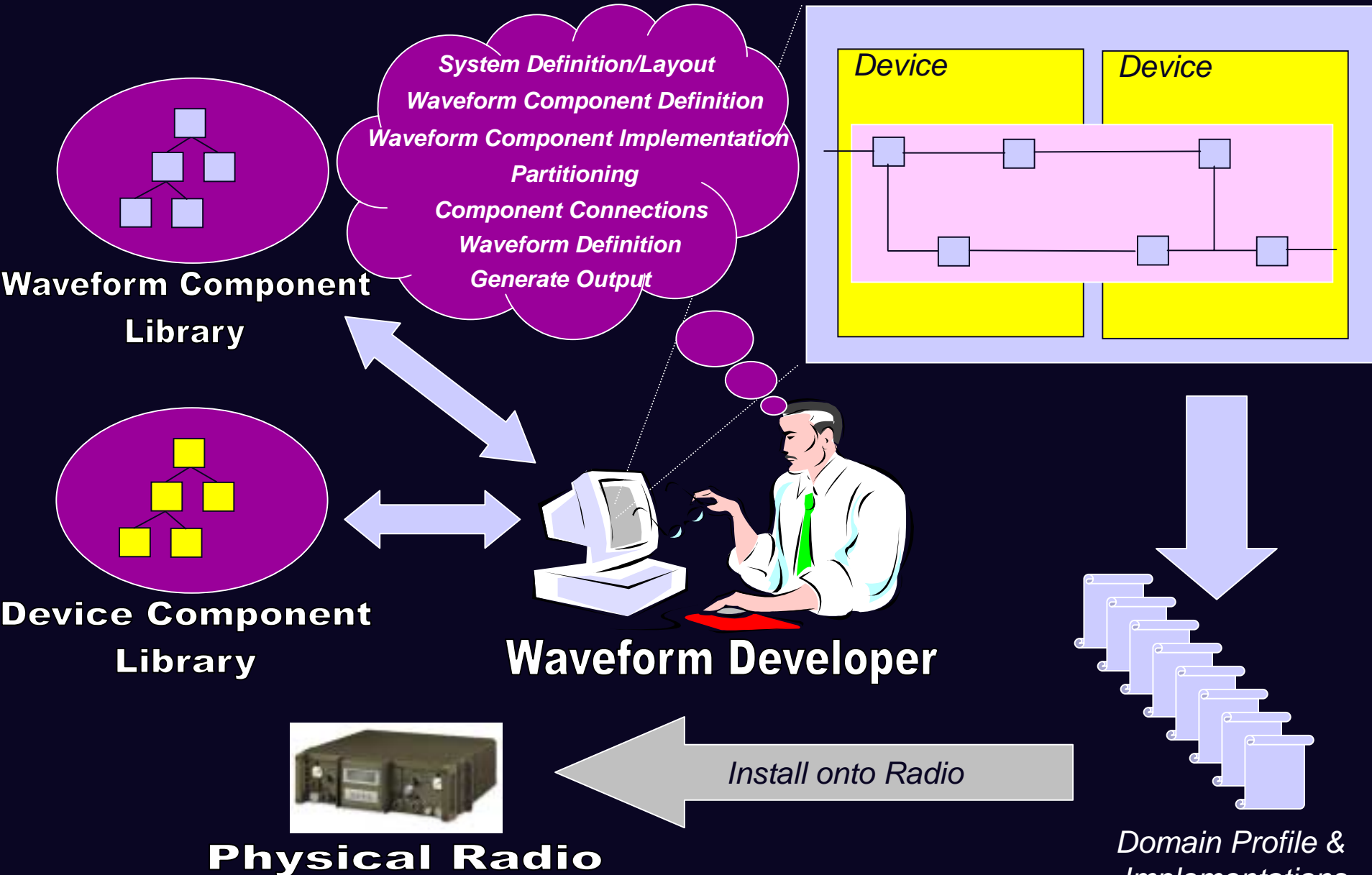
**OMG Embedded and Real-Time Workshop
Reston Va., July 2004**

**Dominick Paniscotti
Bruce Trask**

The SCA vision

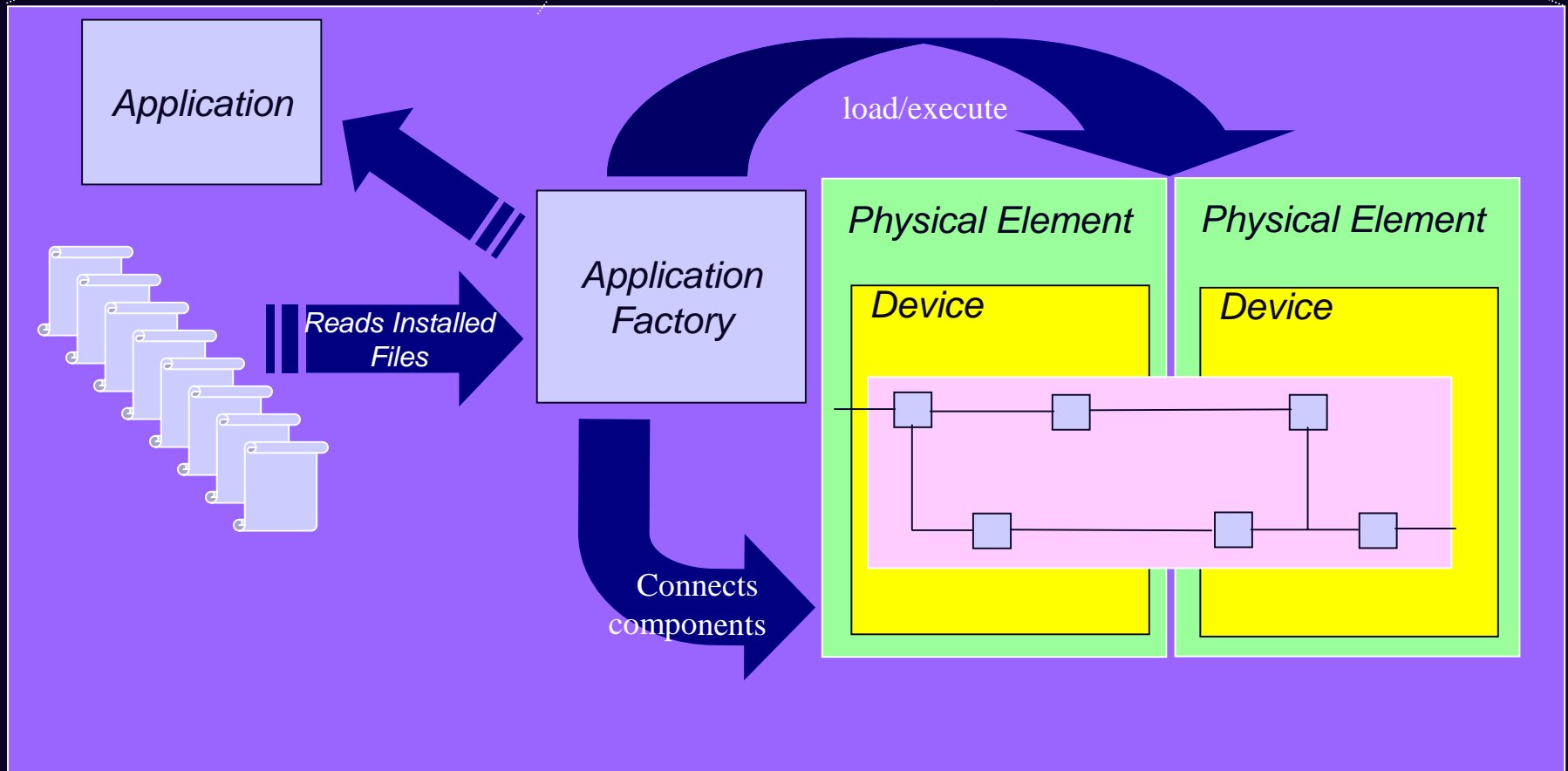
- **The problems:**
 - **Provide portability of application software across diverse platforms**
 - **Support the dynamic deployment and configuration of radio applications across disparate physical hardware**
 - **Promote re-use of the radio applications as well as the software modules that make up the application**
- **The solution**
 - **Deliver a component-based framework**
 - **Abstract the physical hardware from the radio applications to increase portability/maintainability**
 - **Define standard mechanisms to describe, package, and deploy application components**
 - **Use middleware to promote language and hardware independence and provide location transparency**

Waveform Development





Physical Radio



SCA Component Design

The Design of CF::Resource

- **The clients of CF::Resource**
 - **Deployment mechanics of the Core Framework**
 - **Control and Configuration Applications**
 - **Other Components within the Application**
 - **Testing and Monitoring Applications**
- **Define the minimum set of generic functionality required by clients of the component**
 - **start(), stop(), initialize(), releaseObject(), configure(), query(), runTest()**
 - **Segregate the interfaces to promote their re-use and decouple client dependencies**
- **The question... How does one extend the interfaces provided by CF::Resource without direct modification of CF::Resource interface itself?**

The Design of CF::Resource

- **The problems...**
 - **The designers of CF::Resource had no a priori knowledge of the interfaces component developers wanted to add to CF::Resource**
 - **Sub-classing is too static a solution**
 - **May require client side re-compilation**
 - **Risk of CF::Resource interface becoming bloated based upon demands of Application developers.**
 - **Bloating ultimately leading to lack of interface cohesion**
- **To avoid these problems, requires a component design which supports evolution, both anticipated and unanticipated**

The Design of CF::Resource

- **Guiding Design Principles**
 - **Favor object composition over class inheritance¹**
 - **Interface Segregation Principle² – provide a mechanism for clients to retrieve only the interfaces they need and not depend on others provided by the component that they don't use.**
 - **Single Responsibility Principle³ - keep related operations together by defining them in a separate interface.**
 - **Open Closed Principle⁴ – extend the functionality of the component without impact to its existing implementation**

1. Gamma et. al. *Design Patterns*. Reading, MA: Addison-Wesley, 1995, p. 20

2. Martin, *Agile Software Development*, Prentice Hall 2003, p. 135

3. Ibid, p. 95

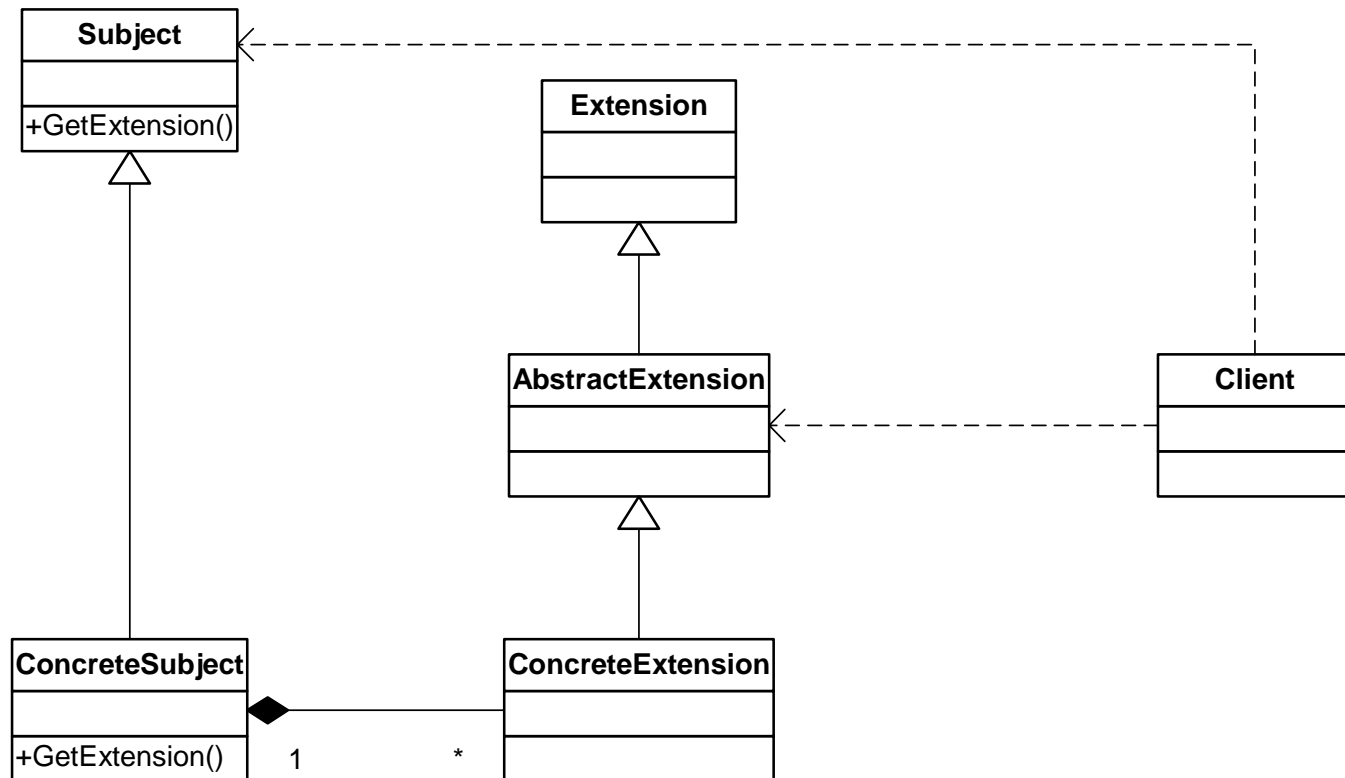
4. Ibid, p. 95

The Design of CF::Resource

- **The Solution: “Export component functionality via *extension interfaces*, one for each semantically-related set of operations”¹**
- **The Patterns: Extension Interface²/Extension Objects³**
- **Apply these patterns to the CF::Resource and the SCA domain**

1. Schmidt, et al. *Pattern-Oriented Software Architecture*, Volume 2, Wiley, 2000, p. 144
2. *ibid* p. 141
3. *Pattern Language of Pattern Design* 3, R. C. Martin, D. Riehle, F. Buschmann (eds.)

Extension Objects Pattern - Structure

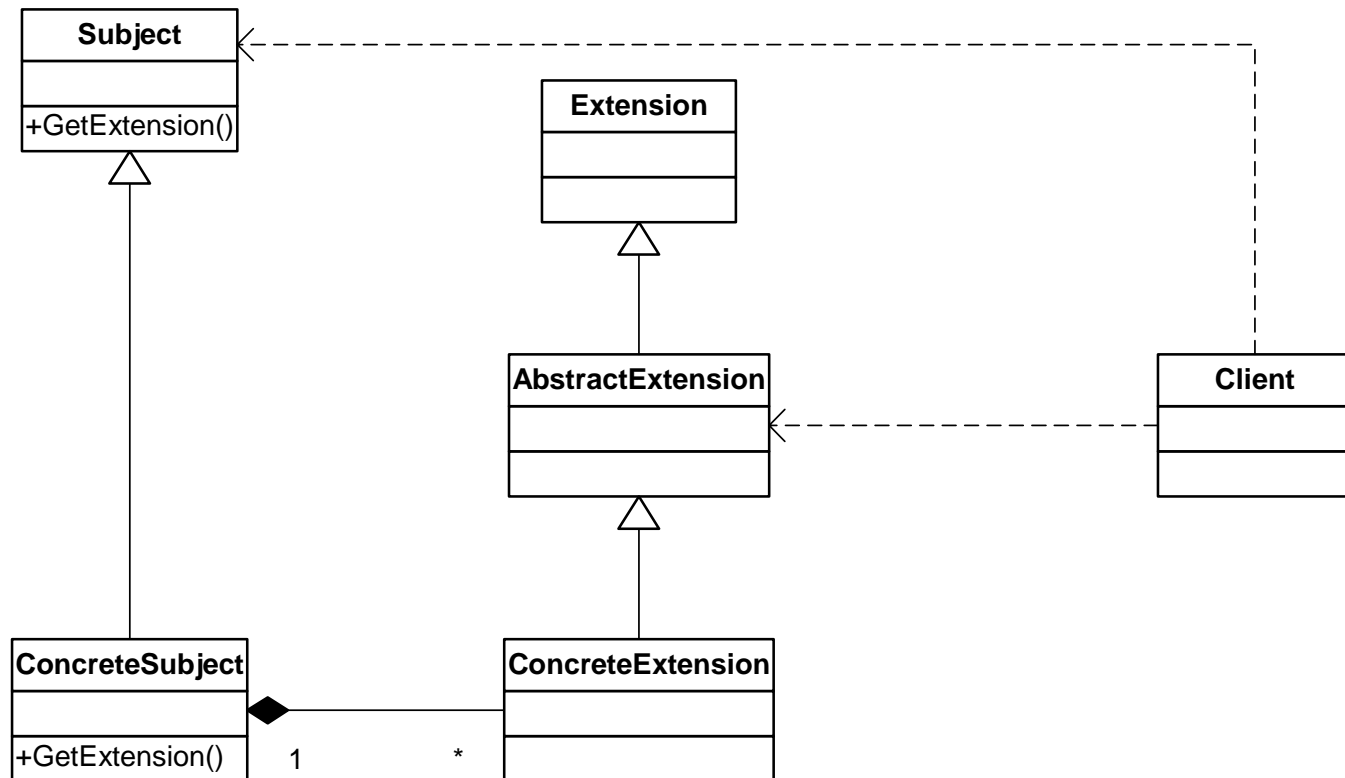


1

- **Subject - Defines the identity of an abstraction. It declares the interface to query whether an object has a particular extension. In the simplest case an interface is identified by a string.²**

1. Pattern Language of Pattern Design 3, R. C. Martin, D. Riehle, F. Buschmann (eds.)
 2. ibid

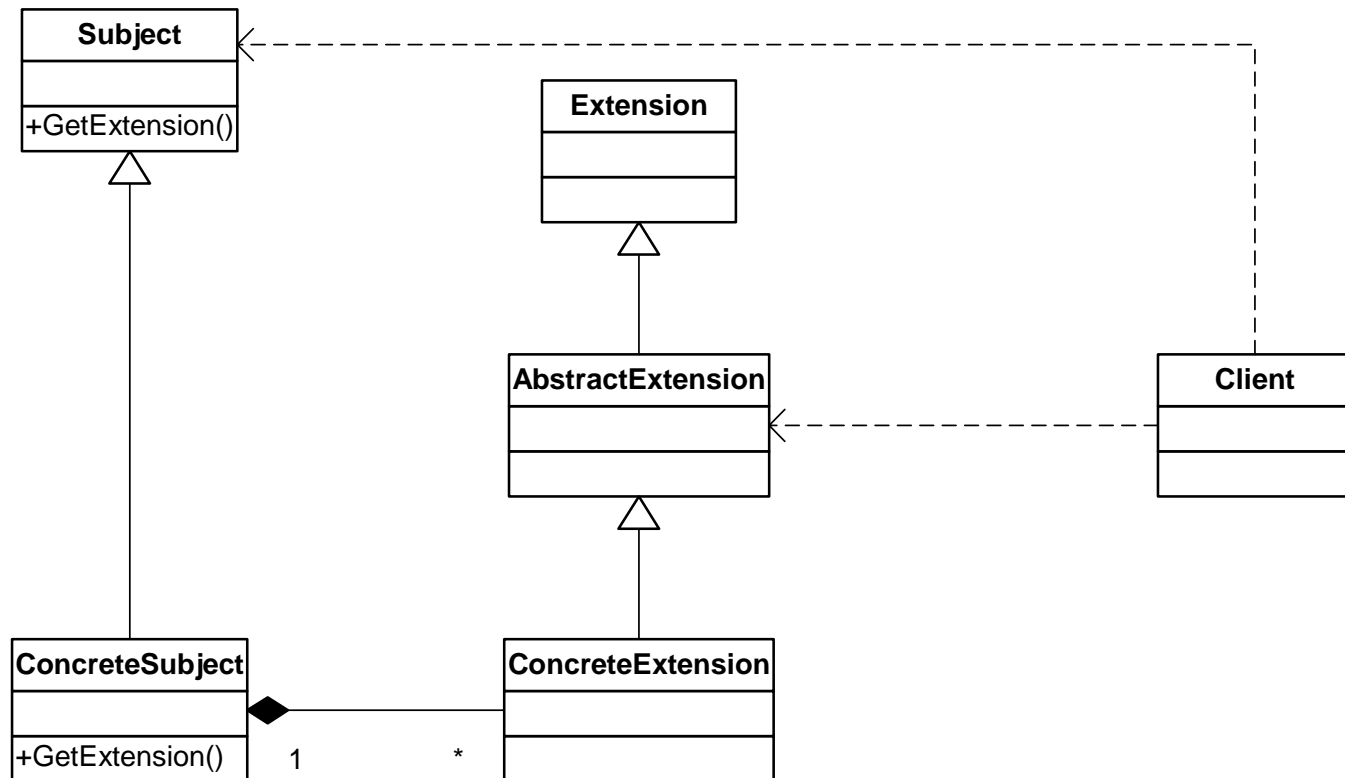
Extension Objects Pattern - Structure



- **Extension** – The base class for all extensions. It defines some support for managing extensions themselves. Extensions knows its owning object²

1. Pattern Language of Pattern Design 3, R. C. Martin, D. Riehle, F. Buschmann (eds.)
2. ibid

Extension Objects Pattern - Structure

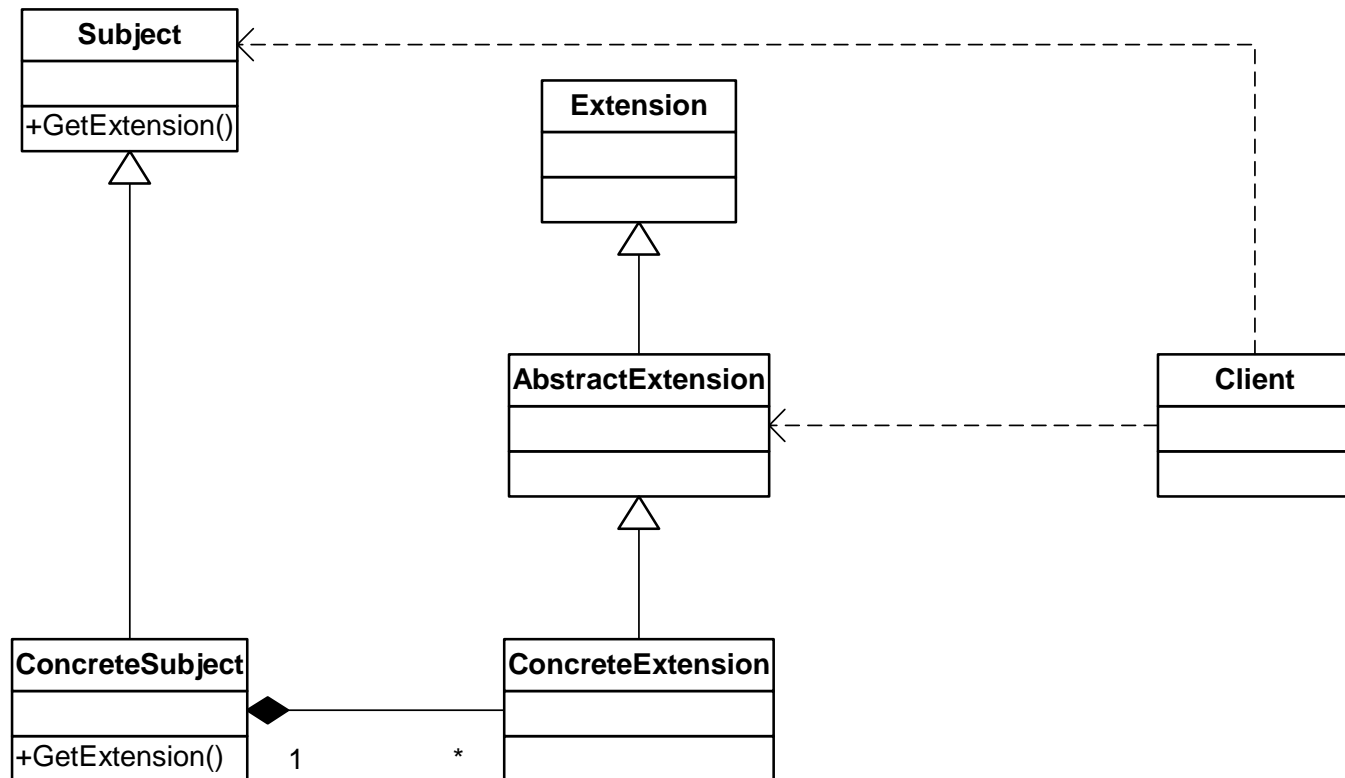


1

- **ConcreteSubject** – Implement the **GetExtension** operation to return a corresponding extension object when the client asks for it²

1. Pattern Language of Pattern Design 3, R. C. Martin, D. Riehle, F. Buschmann (eds.)
 2. ibid

Extension Objects Pattern - Structure

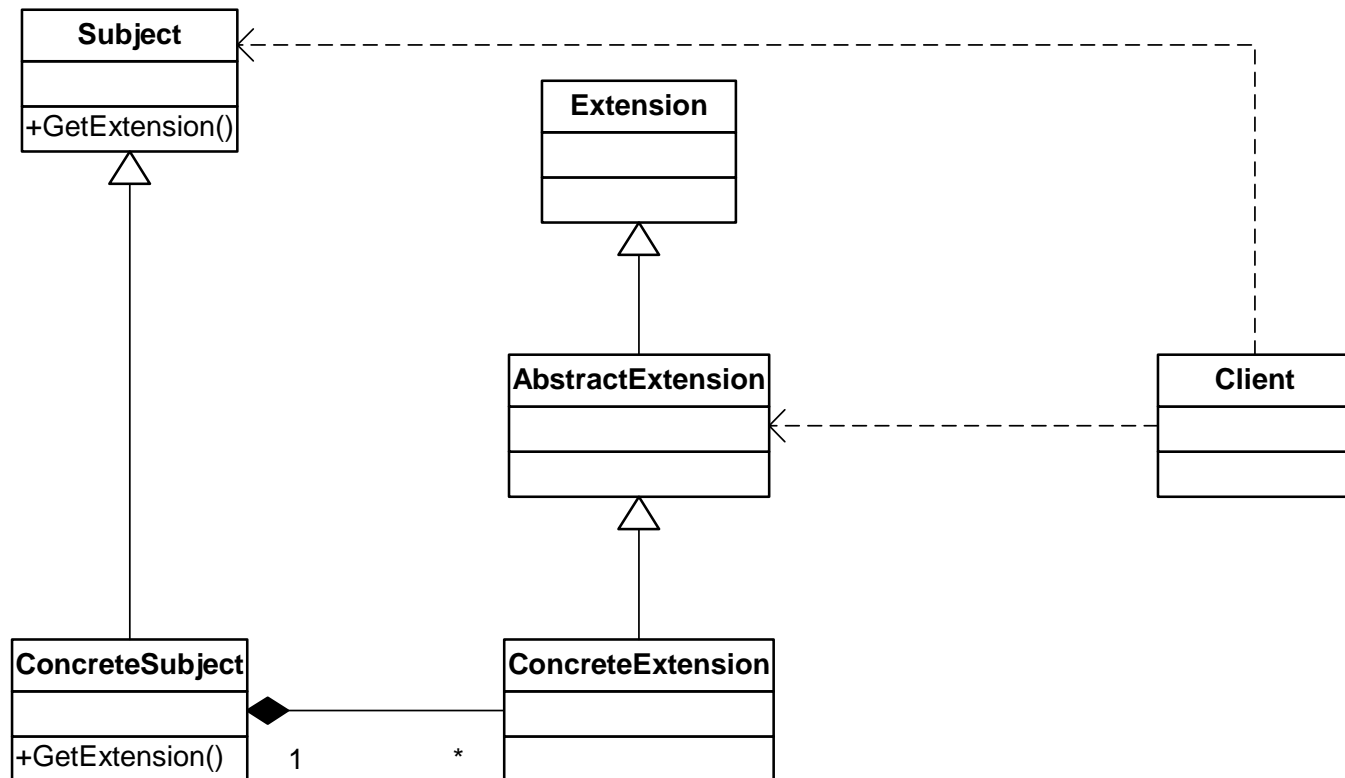


1

- **AbstractExtension – Declares the interface for a specific extension²**

1. Pattern Language of Pattern Design 3, R. C. Martin, D. Riehle, F. Buschmann (eds.)
 2. ibid

Extension Objects Pattern - Structure

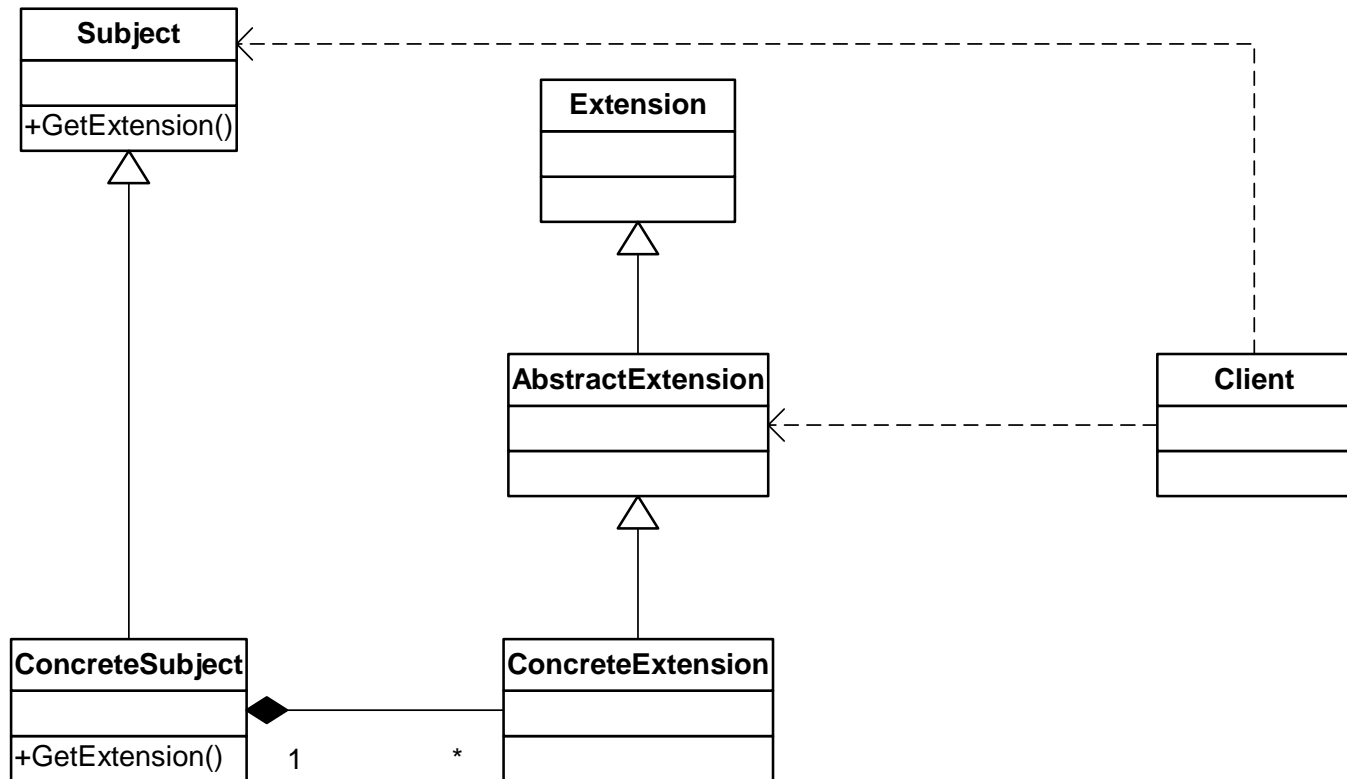


1

- **ConcreteExtension – Implement the extension interface for a particular component. Store the state associated with a specific extension²**

1. Pattern Language of Pattern Design 3, R. C. Martin, D. Riehle, F. Buschmann (eds.)
 2. ibid

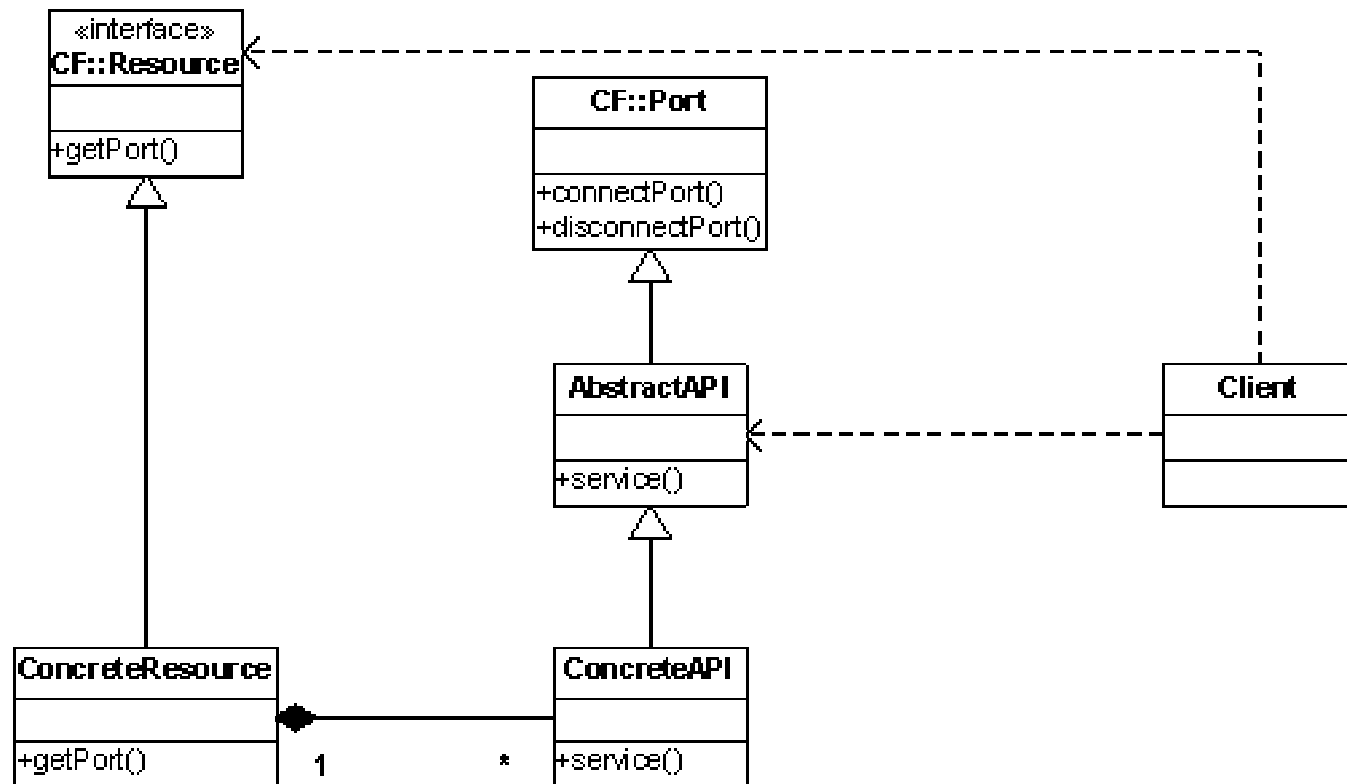
Extension Objects Pattern (applied)



1

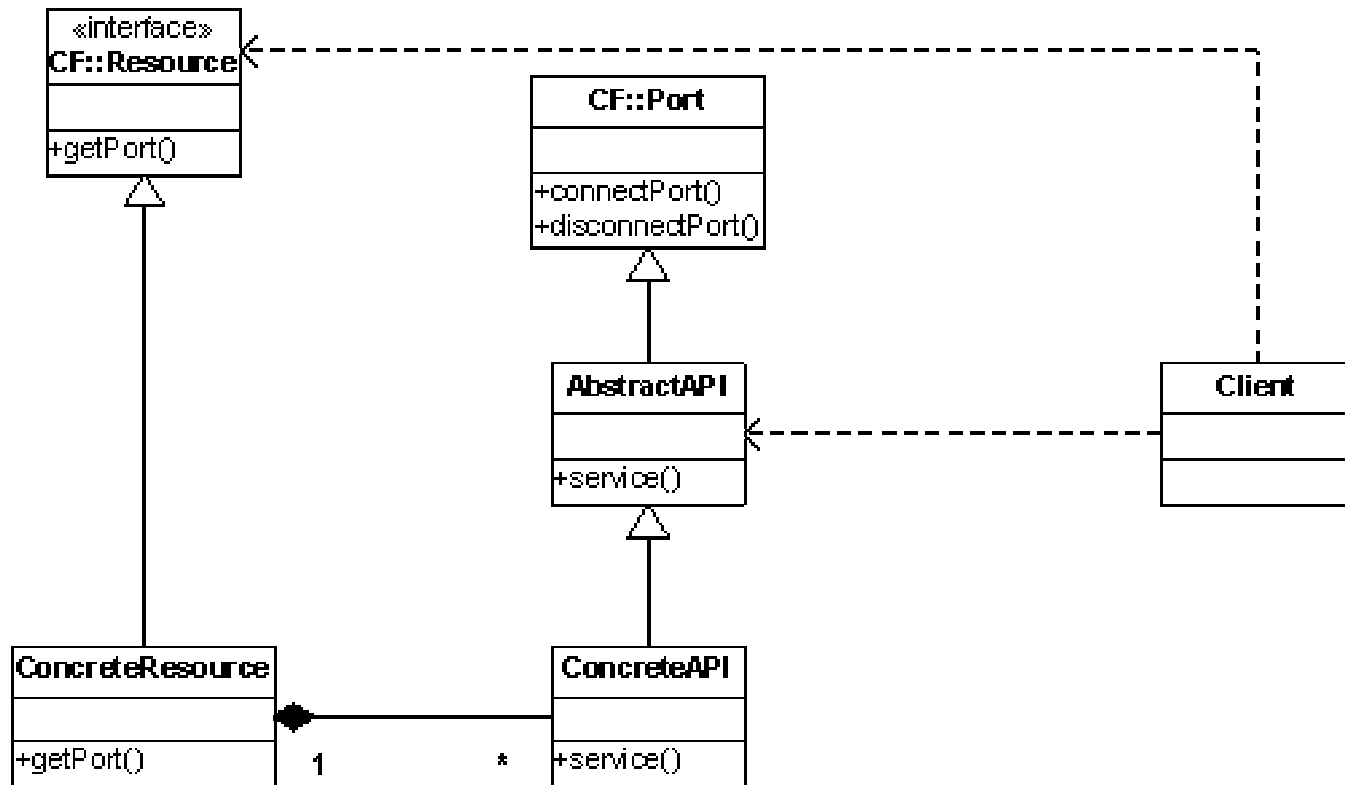
1. Pattern Language of Pattern Design 3, R. C. Martin, D. Riehle, F. Buschmann (eds.)

Extension Objects Pattern (applied)



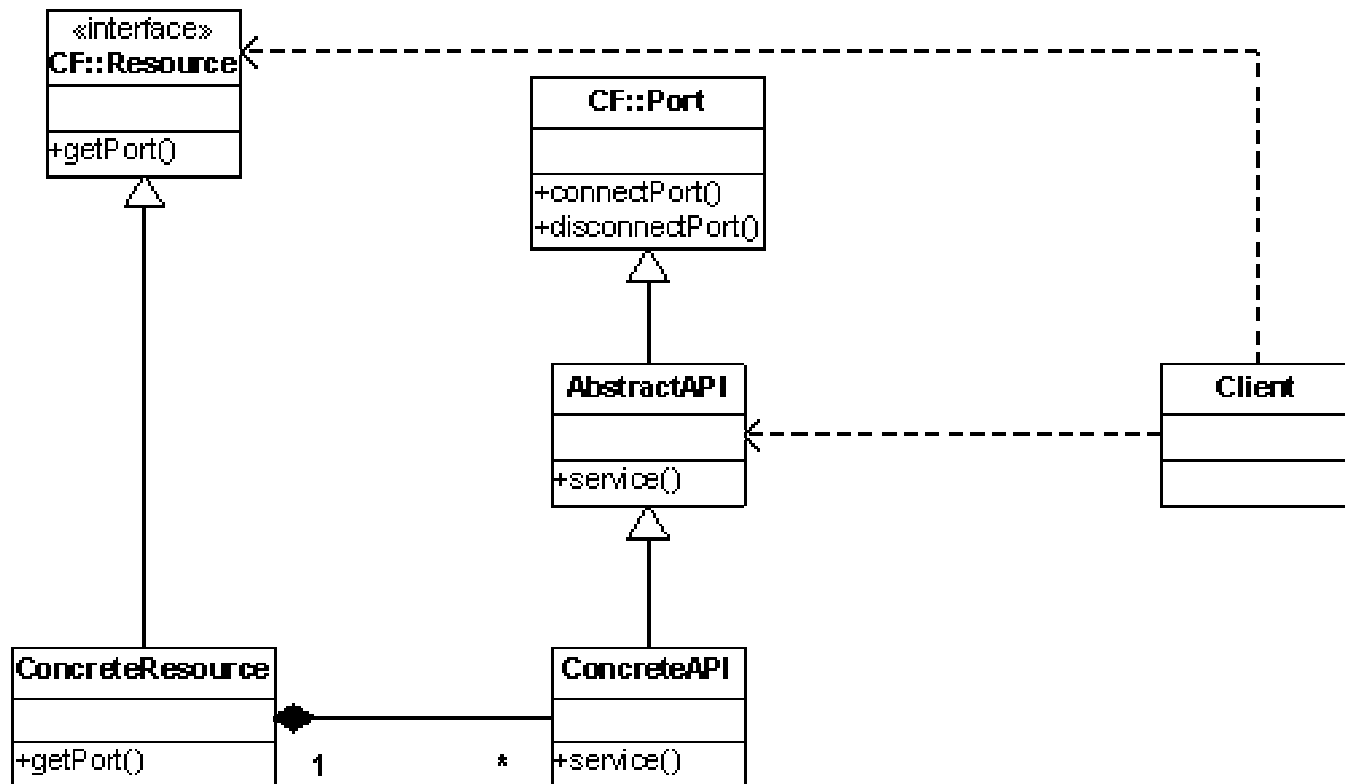
- Almost a one to one mapping of the Extension Object pattern to the SCA Domain

Extension Objects Pattern (applied)



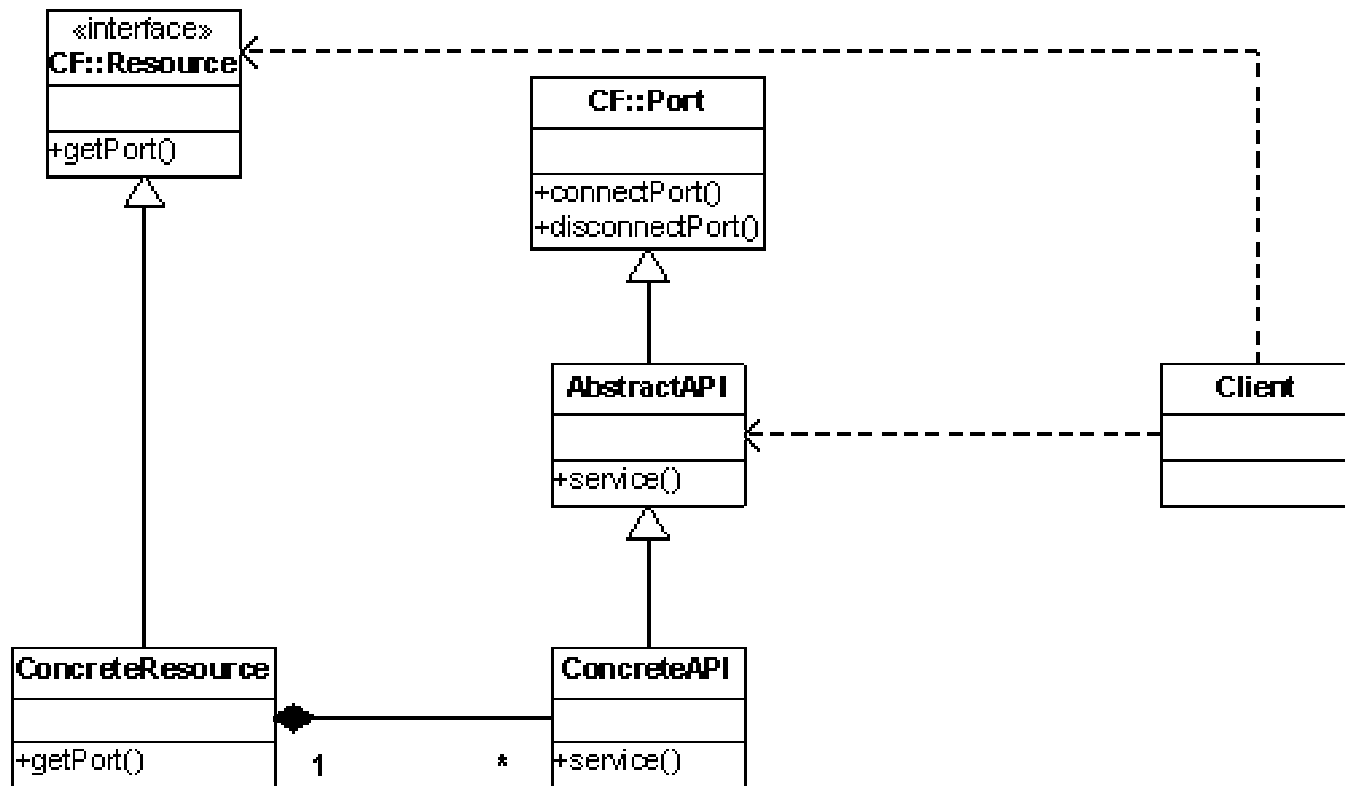
- **CF::Resource** – the identity of radio waveform and radio platform components. Declares `getPort` method for client to query if an `CF::Resource` supports a particular interface (i.e. extension)

Extension Objects Pattern (applied)



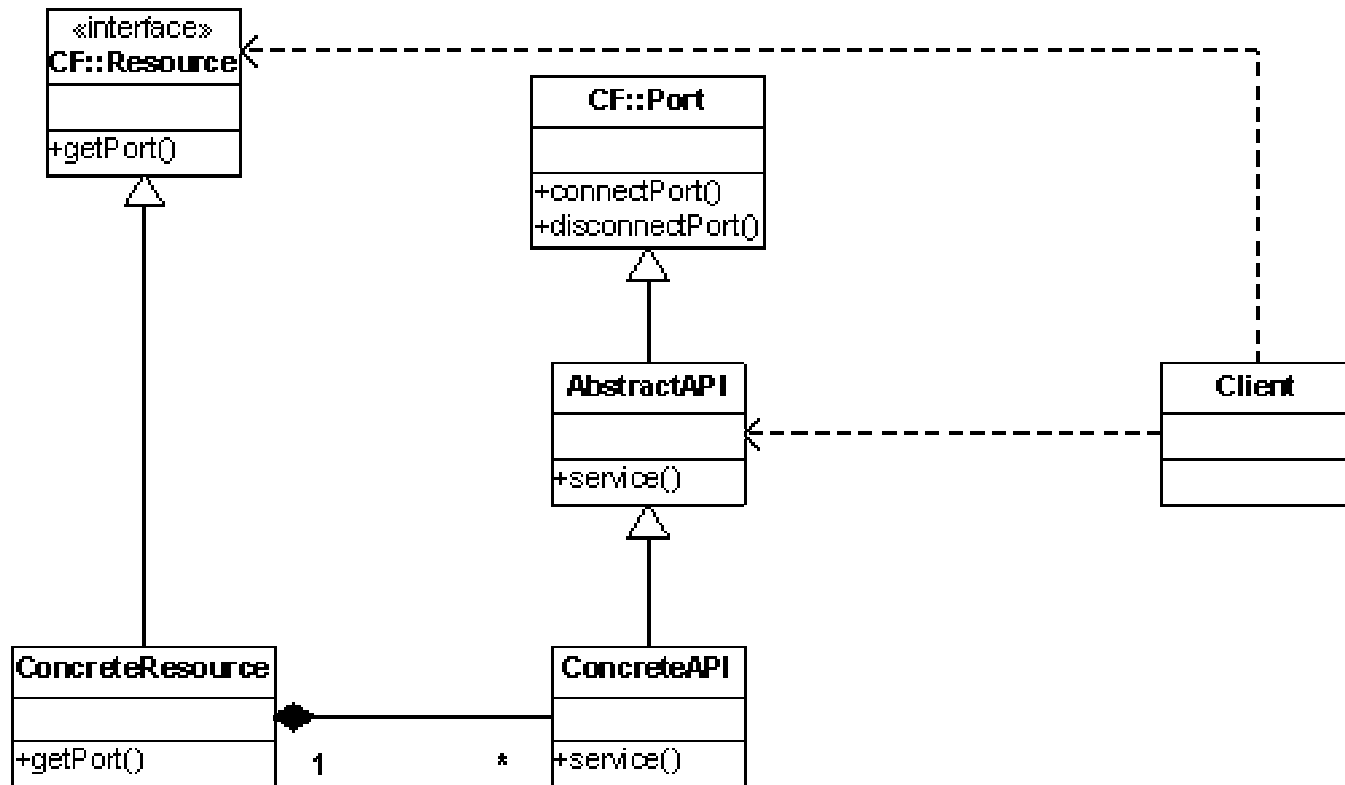
- **CF::Port** – Base class for all “producer” or “uses” extensions. It defines support for managing “connections” via `connectPort()` and `disconnectPort()`

Extension Objects Pattern (applied)



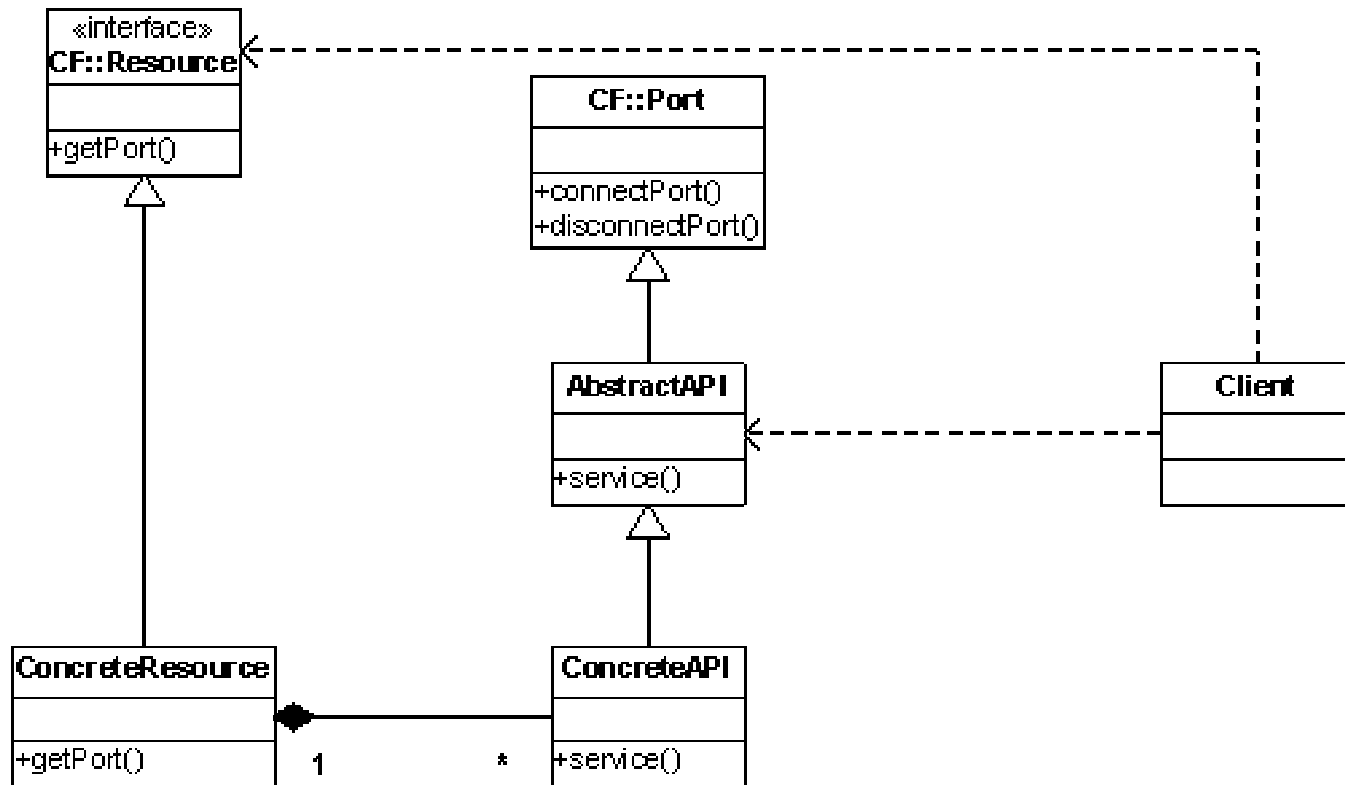
- **ConcreteResource** – implements the `getPort` method to return the corresponding port (extension object) when the client asks for it

Extension Objects Pattern (applied)



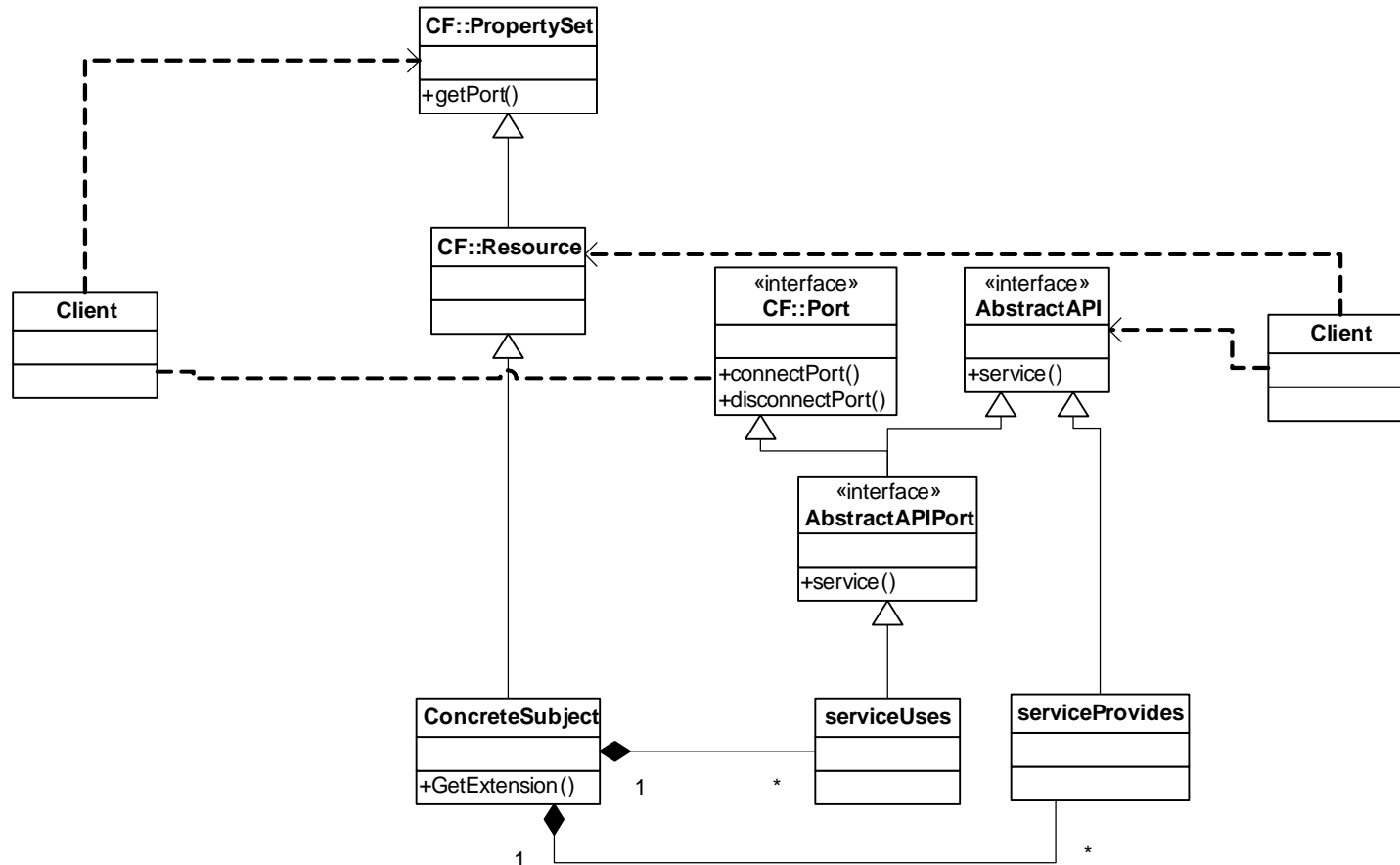
- **AbstractAPI** – declares the interface for the particular API, usually from the SCA API Supplement (e.g. Packet BB)

Extension Objects Pattern (applied)



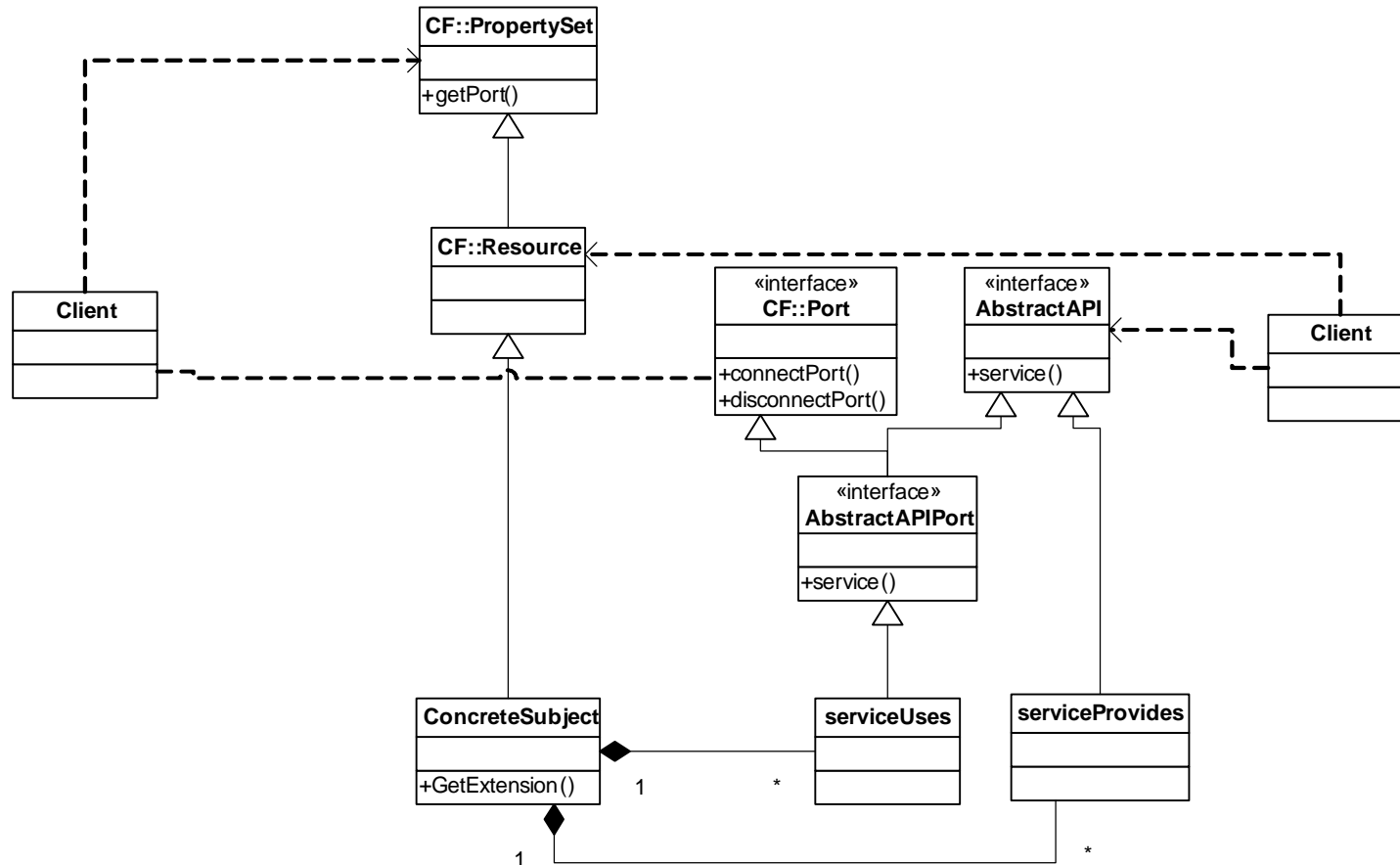
- **ConcreteAPI** – Implements the **AbstractAPI** for a particular component.

Extension Object Pattern – Variants for the SCA



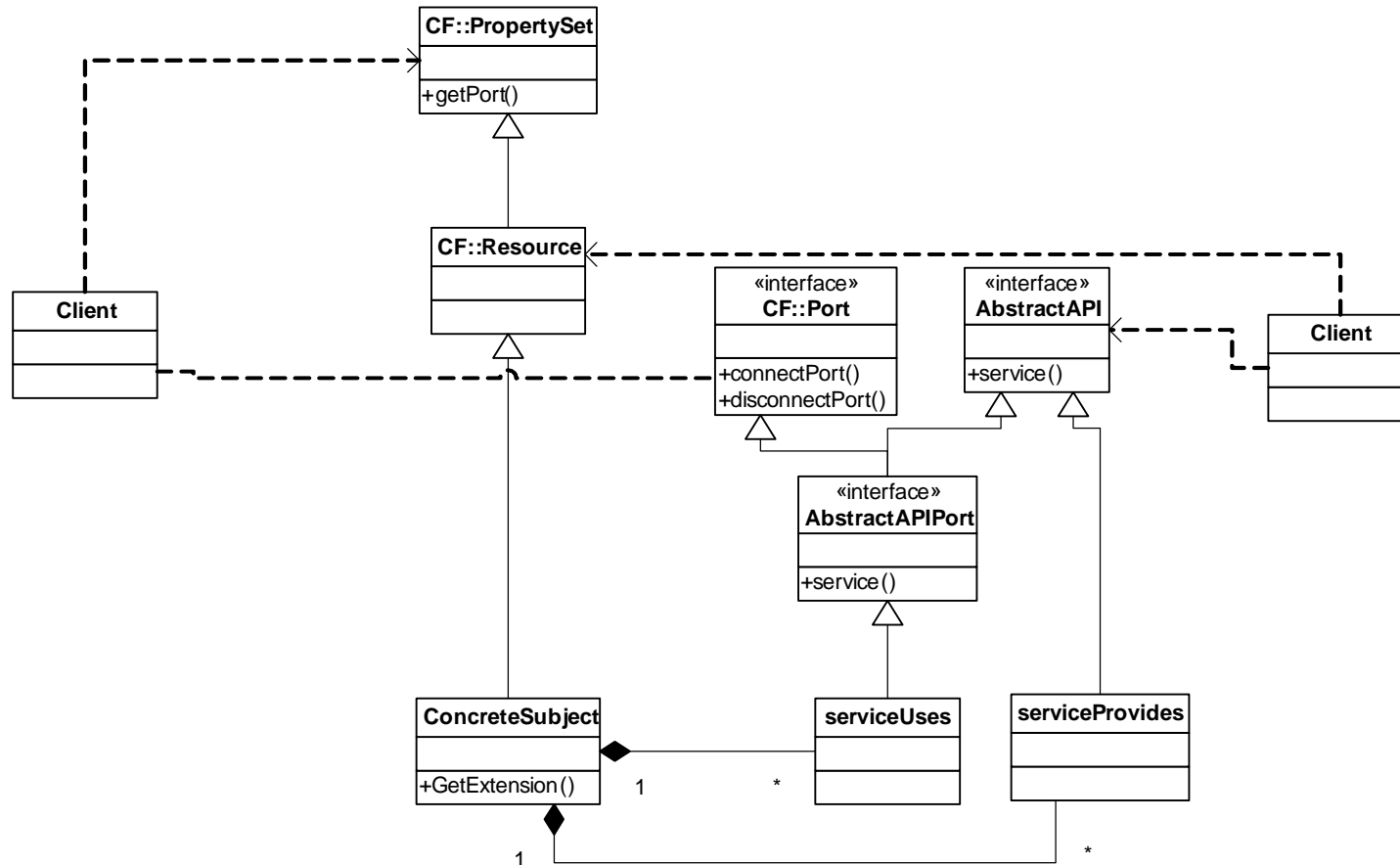
- **Move the `getPort` method into a separate base interface so that it can be reused elsewhere in the SCA and also decouple the clients of the `getPort` call from other **CF::Resource** operations**

Extension Object Pattern – Variants for the SCA



- **Remove inheritance relationship between CF::Port and AbstractAPI and replace it with an empty interface that is a mix in of CF::Port and AbstractAPI.**

Extension Object Pattern – Variants for the SCA

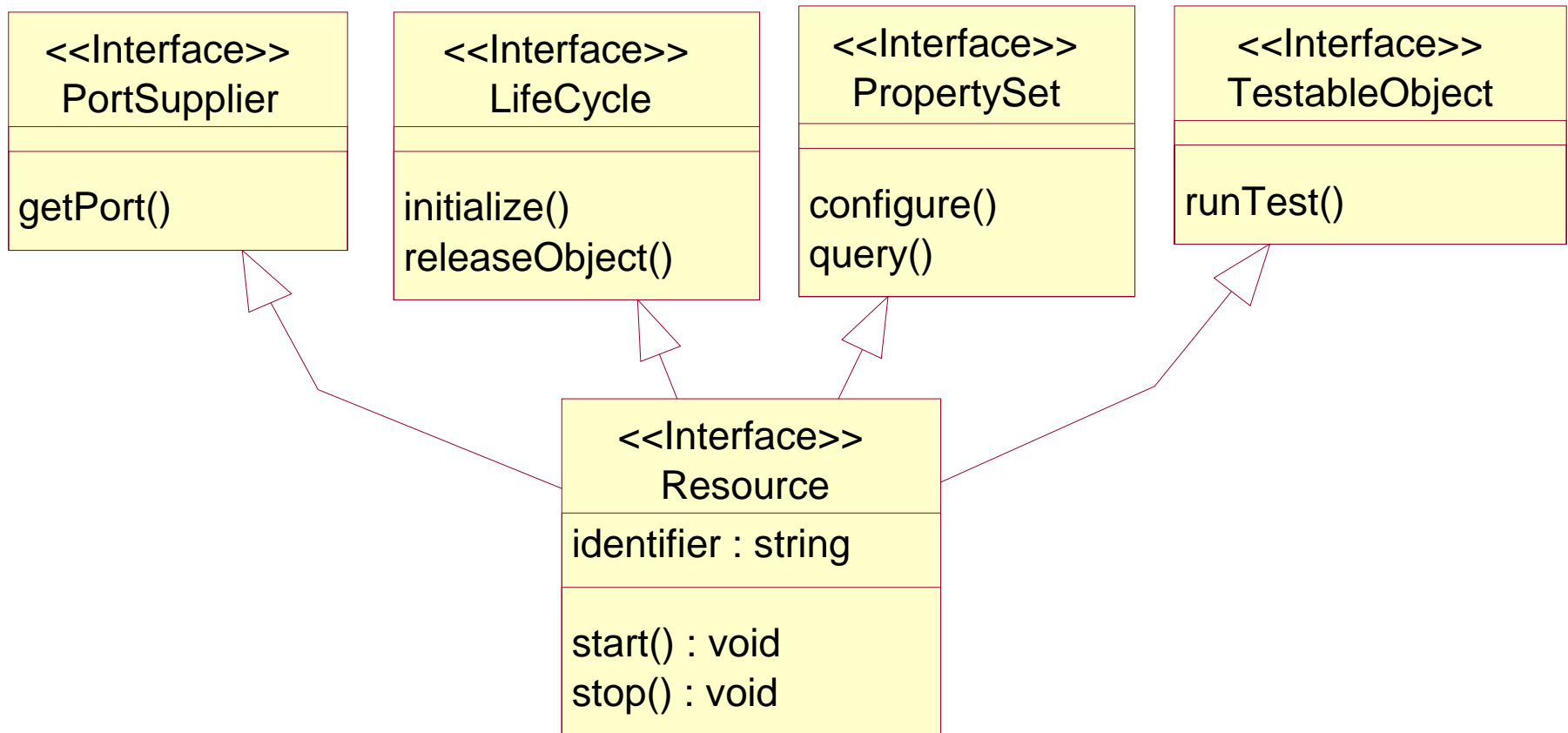


- ServiceUses/ServiceProvides: Separate the interfaces of the concrete API realization into “uses” and “provides” implementations. Notice “provides” side does not inherit the CF::Port interface as connections are “one way”. They simply offer the service to the user client.**

Extension Objects Pattern – Benefits Summary

- **Maintain the CF::Resource and CF::Device abstractions while simultaneously providing for unanticipated “interface” extensions.**
- **Keeps clients of waveforms decoupled from various interfaces that they don't use**
- **Allows for easy addition of new services inside components with a minimal impact to client code**
- **Prevents bloated interfaces**

CF::Resource Interface



SCA Component usage

Component Interconnectivity

- **The question: How does one effectively connect components that have extended their interfaces via the extension objects pattern?**
- **The problems:**
 - **Component-based infrastructures introduce a slightly different use case than typical extension object usage.**
 - **The extension interfaces are used to support inter-component communication**
 - **Components may have multiple, disparate connections to multiple other components or to the same component.**
 - **How does one change the “schematic” of the component assembly with a minimum of impact?**
 - **Can we achieve functionality similar to wiring ICs together?**
 - **E.g. fan-out. Constraints to ensure proper output pins getting connected to the proper input pins.**
 - **Connection related functionality can become a burdensome responsibility for the component itself as more and more connections are added.**

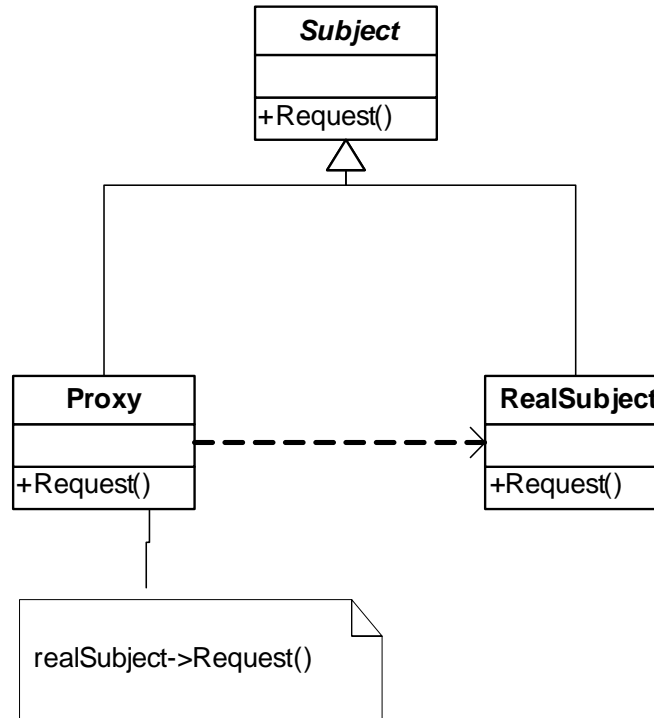
Component Interconnectivity

- **The guiding principles:**
 - **Good to separate the business logic of the component from the inter-component communication mechanics.**
 - **Decouple axes of change and functionality into separate classes**
 - **Prefer object composition over class inheritance**
 - **Decouple the functionality used by one client from that used by others**
- **The patterns:**
 - **Proxy¹, Observer²**
- **The solution:**
 - **Create a unique producer port for each logical connection**
 - **Connect this producer to a consumer**
 - **Provide user-defined callbacks for to transfer the consumer call to the component itself.**

1. Gamma et. al. *Design Patterns*. Reading, MA: Addison-Wesley, 1995, p. 207

2. *Ibid*, p. 293

The Proxy Pattern - Structure



1

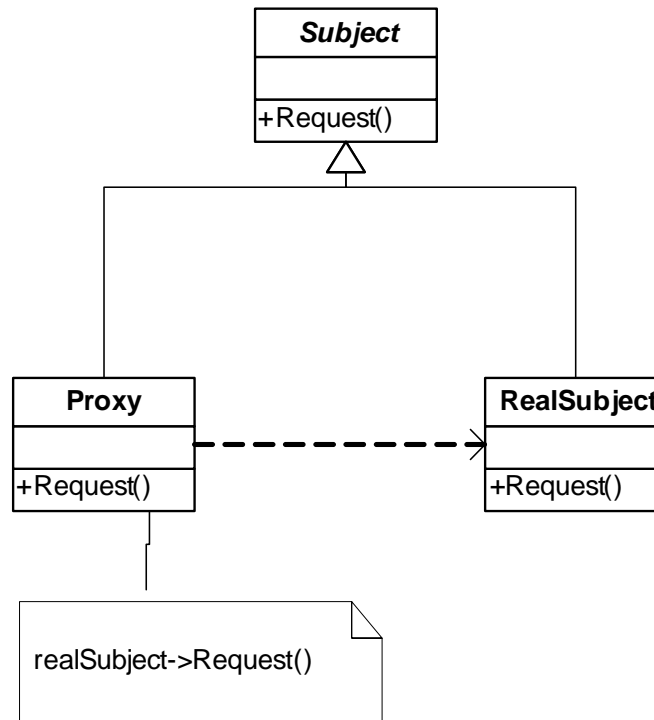
- **Proxy**

- “maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same”
- “Provides an interface identical to Subject’s so that a proxy can be substituted for the real subject”²

1. Gamma et. al. *Design Patterns*. Reading, MA: Addison-Wesley, 1995, p. 209

2. Ibid p.209 *BAE SYSTEMS Information and Electronic Systems Integration Inc.*

The Proxy Pattern - Structure



1

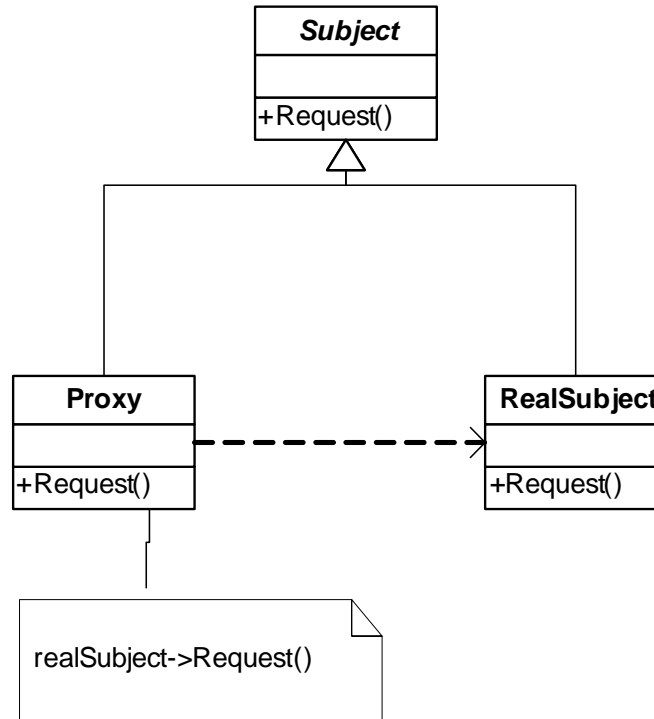
- **Proxy**

- “Controls access to the real subject and may be responsible for creating and deleting it”²
- Other responsibilities include being a “remote proxy”, “virtual proxy” etc.

1. Gamma et. al. *Design Patterns*. Reading, MA: Addison-Wesley, 1995, p. 209

2. Ibid p. 209 *BAE SYSTEMS Information and Electronic Systems Integration Inc.*

The Proxy Pattern - Structure



1

- **Subject**

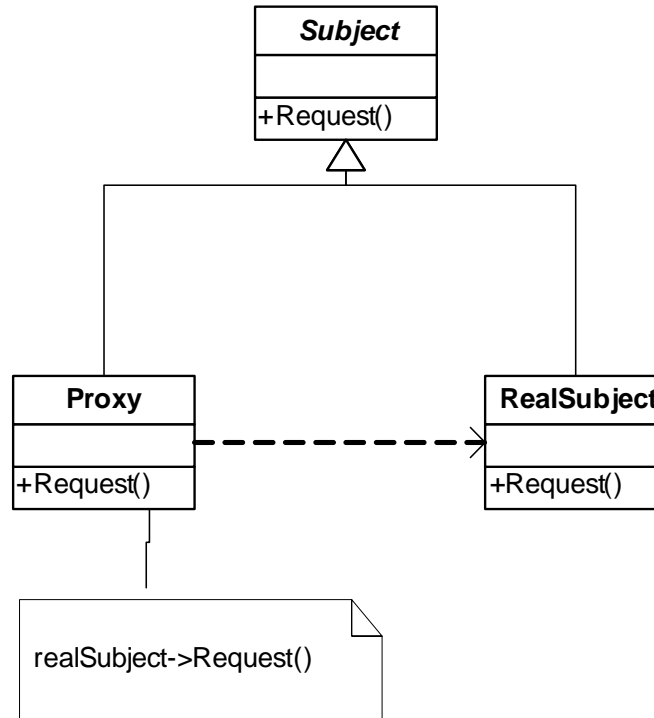
- “Defines the common interface for **RealSubject** and **Proxy** so that a **Proxy** can be used anywhere a **RealSubject** is expected”²

1. Gamma et. al. *Design Patterns*. Reading, MA: Addison-Wesley, 1995, p. 209

2. Ibid p.210 *BAE SYSTEMS Information and Electronic Systems Integration Inc.*

Communication, Navigation, Identification and Reconnaissance (CNIR) - Copyright 2004

The Proxy Pattern - Structure



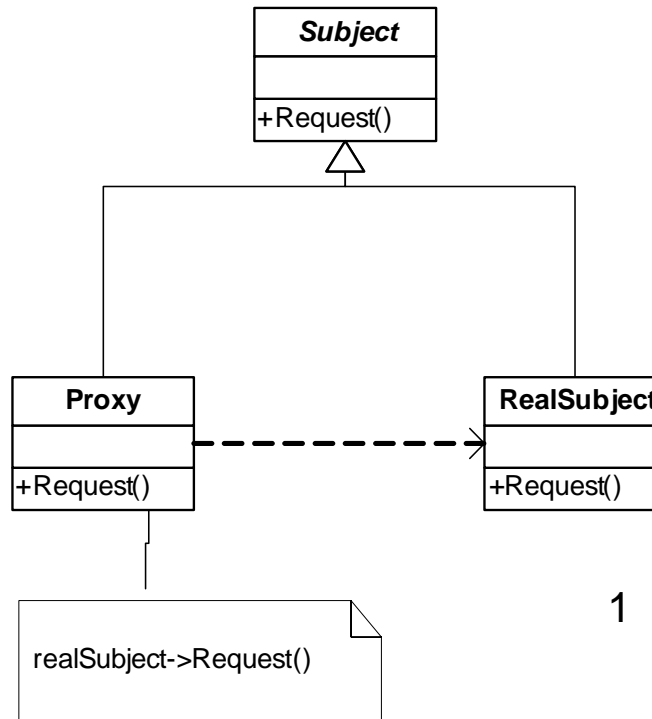
1

- **RealSubject**

- “Defines the real object that the proxy represents”²

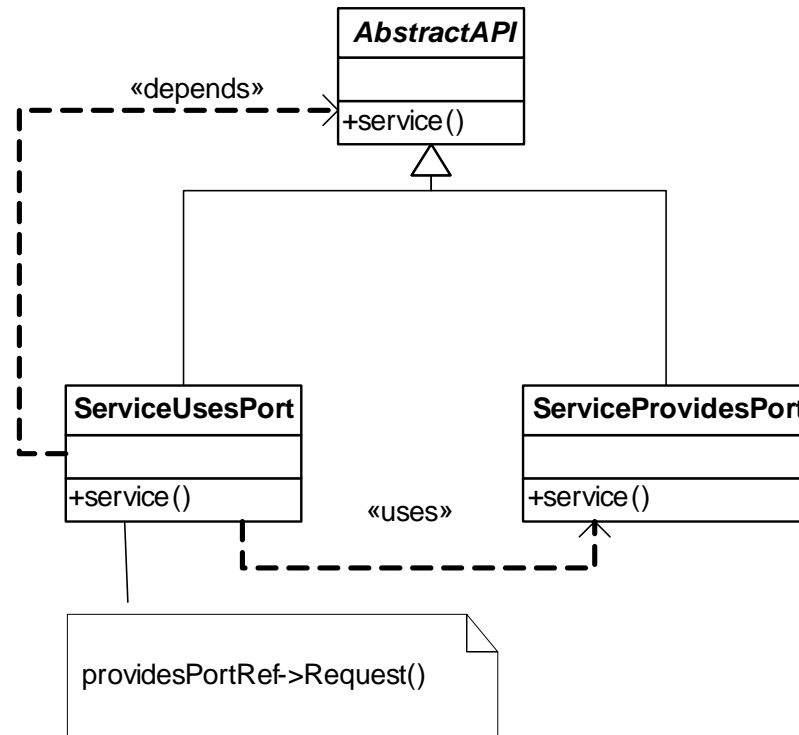
1. Gamma et. al. *Design Patterns*. Reading, MA: Addison-Wesley, 1995, p. 209
2. Ibid, p.210 *BAE SYSTEMS Information and Electronic Systems Integration Inc. Communication, Navigation, Identification and Reconnaissance (CNIR) - Copyright 2004*

Mapping the Proxy Pattern to the SCA Domain



1. Gamma et. al. *Design Patterns*. Reading, MA: Addison-Wesley, 1995, p. 209

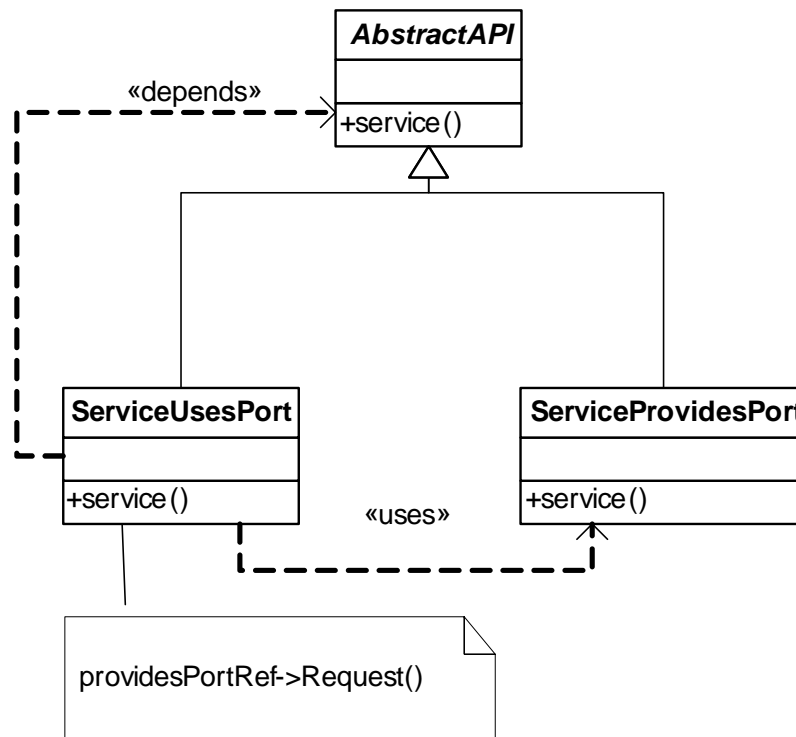
Mapping the Proxy Pattern to the SCA Domain



- **AbstractAPI**

- Typically this is one of the APIs from the SCA API Supplement. E.g PushPacket.

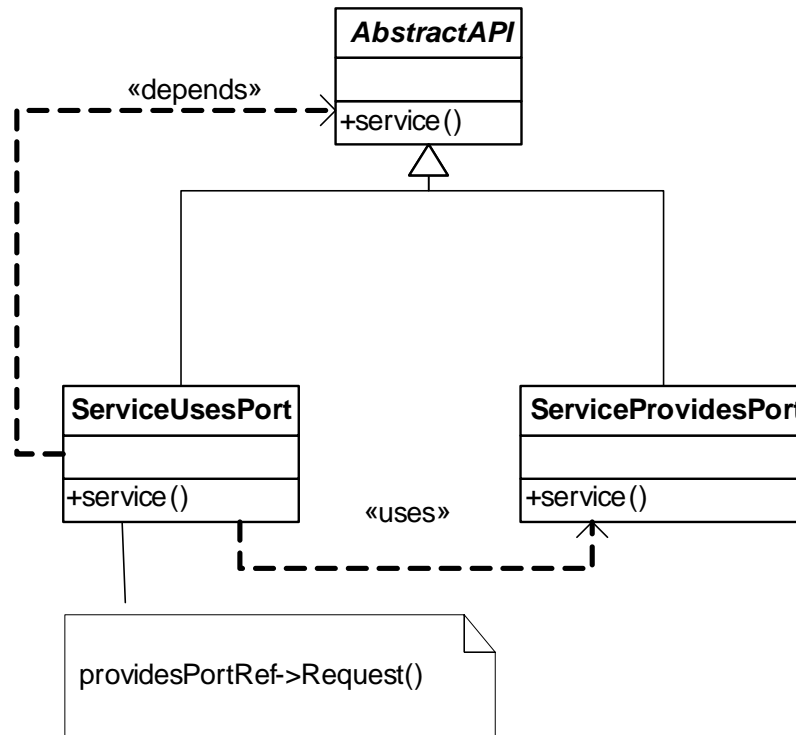
Mapping the Proxy Pattern to the SCA Domain



- **ServiceUsesPort**

- The “usesport” in the SCA. A.k.a producer ports, supplier ports, publishing ports. Responsible for disseminating calls to “providesports”.

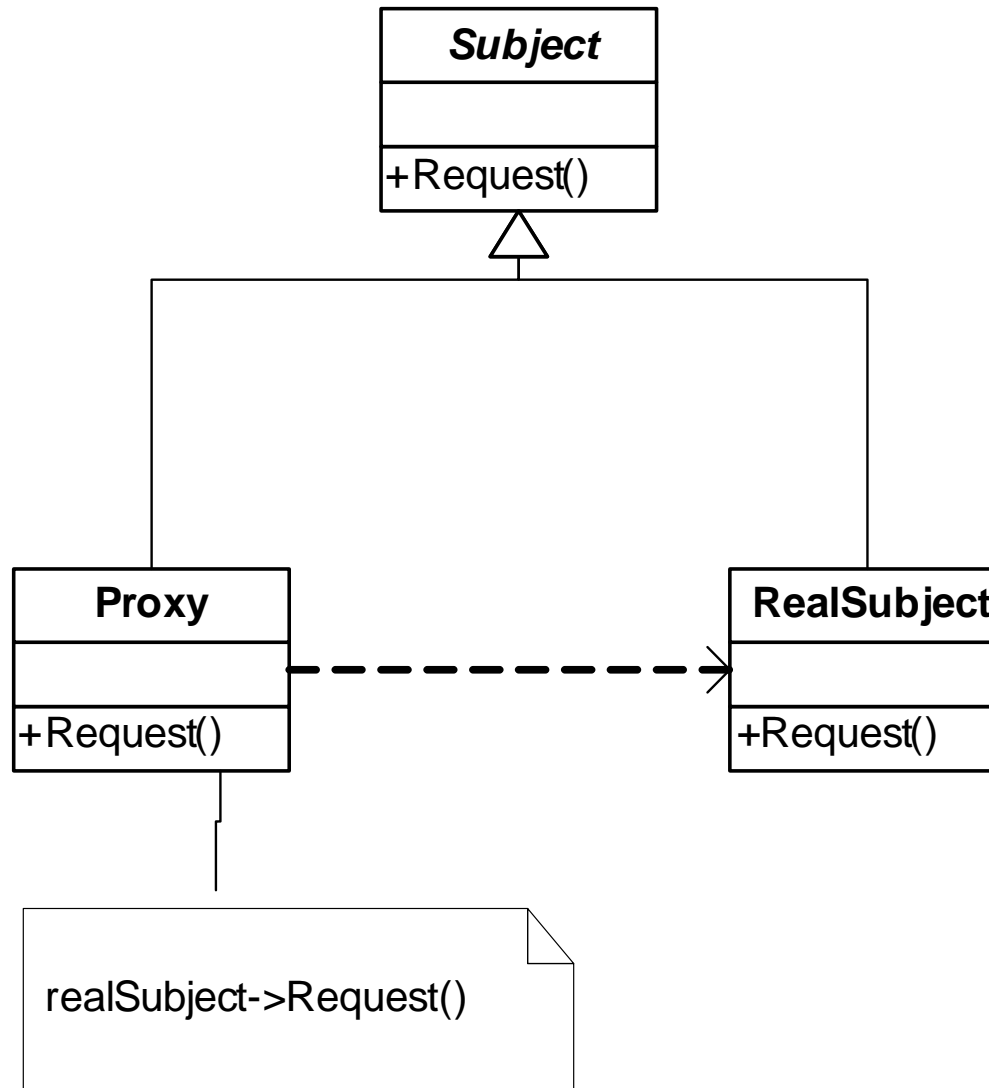
Mapping the Proxy Pattern to the SCA Domain



- **ServiceProvidesPort**

- The “providesport” in the SCA. A.k.a consumer ports, subscriber ports. Responsible for providing a the service API and bridging to the receiving component.

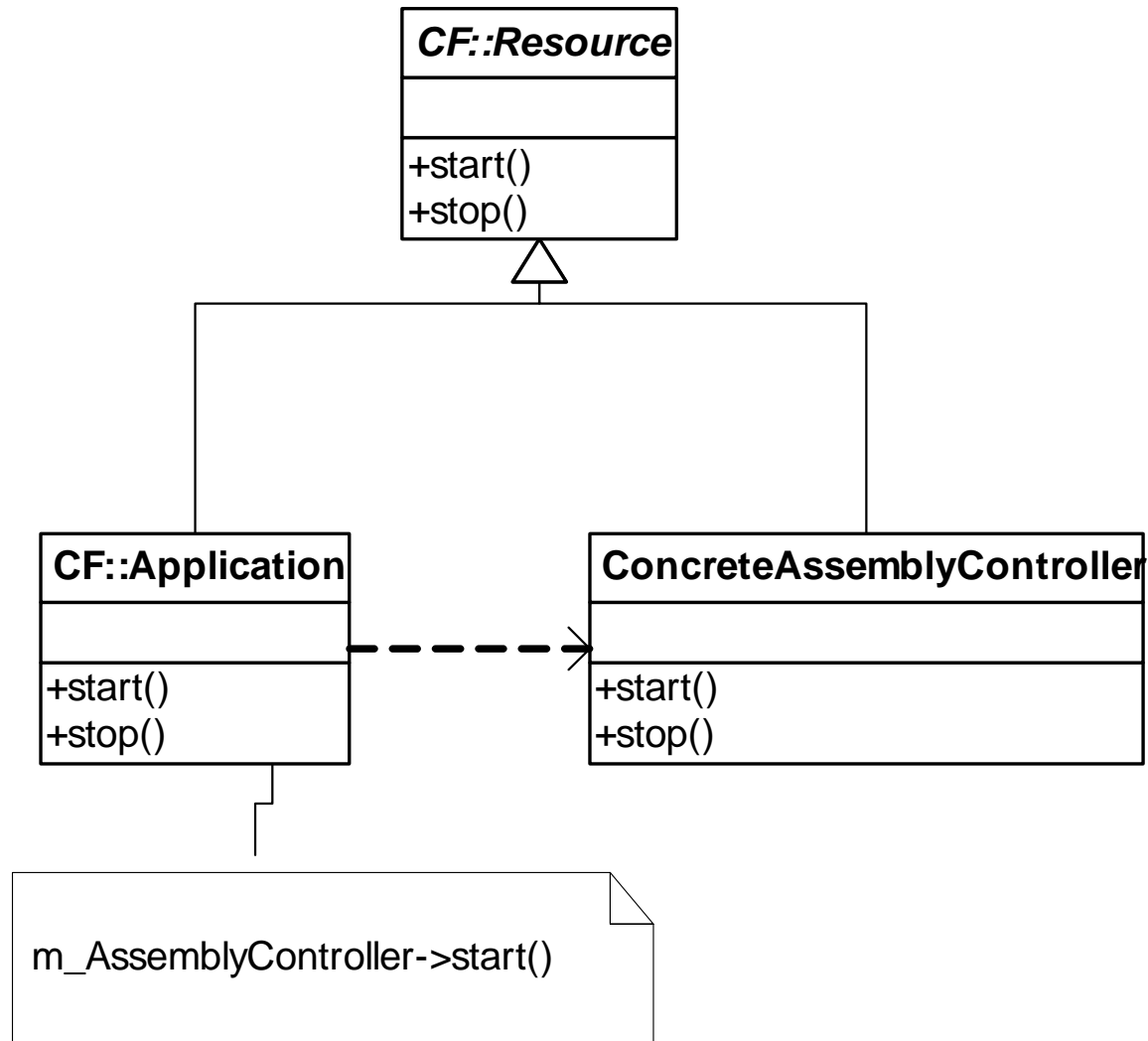
Proxy Pattern continued



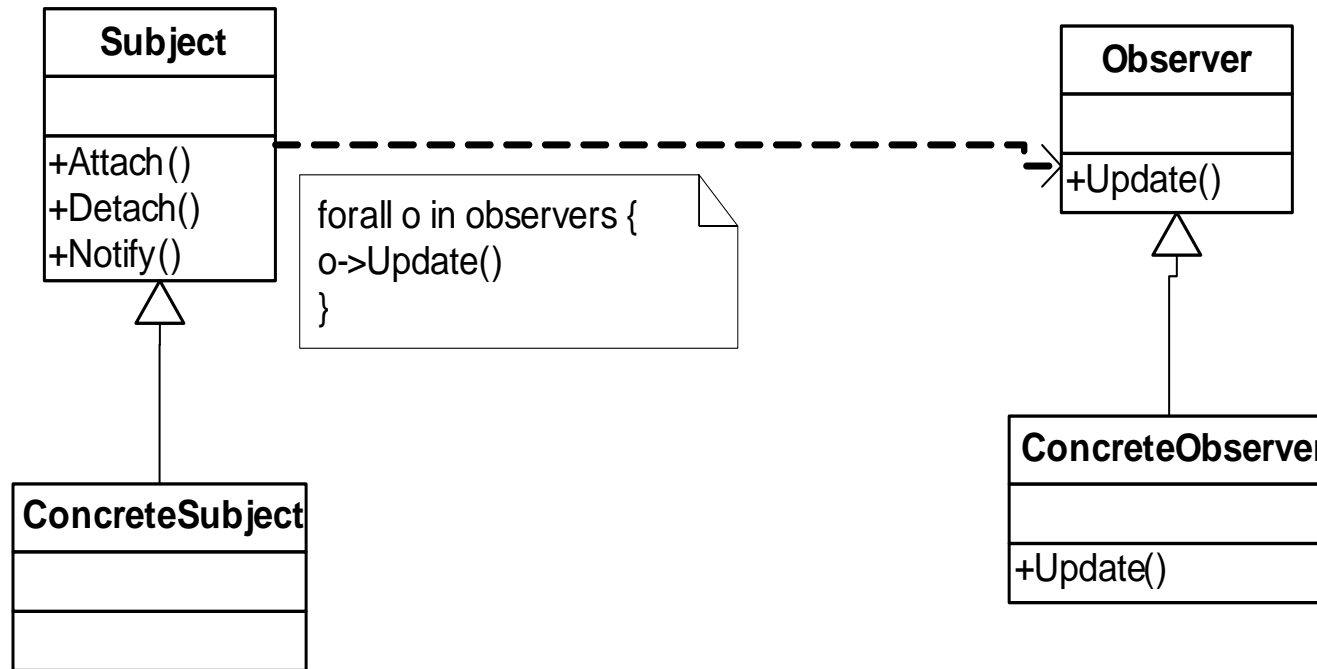
1

1. Gamma et. al. *Design Patterns*. Reading, MA: Addison-Wesley, 1995, p. 209

Proxy Pattern continued



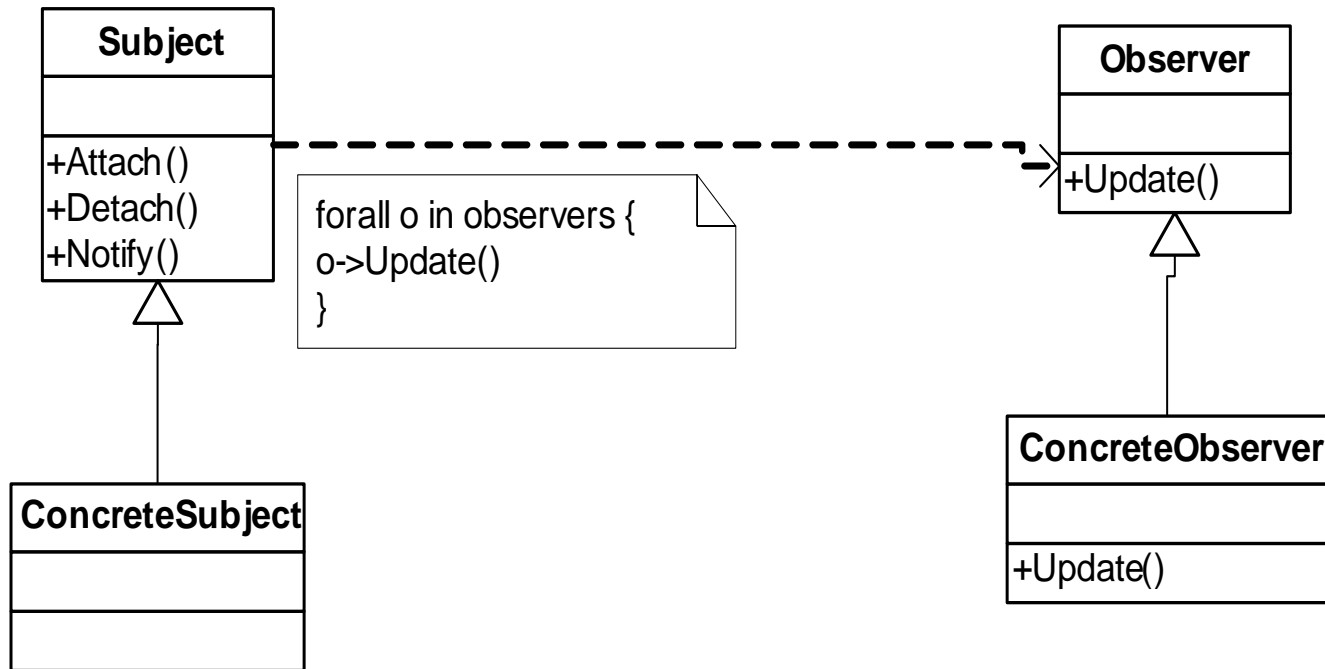
The Observer Pattern



- Intent – “Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically”²

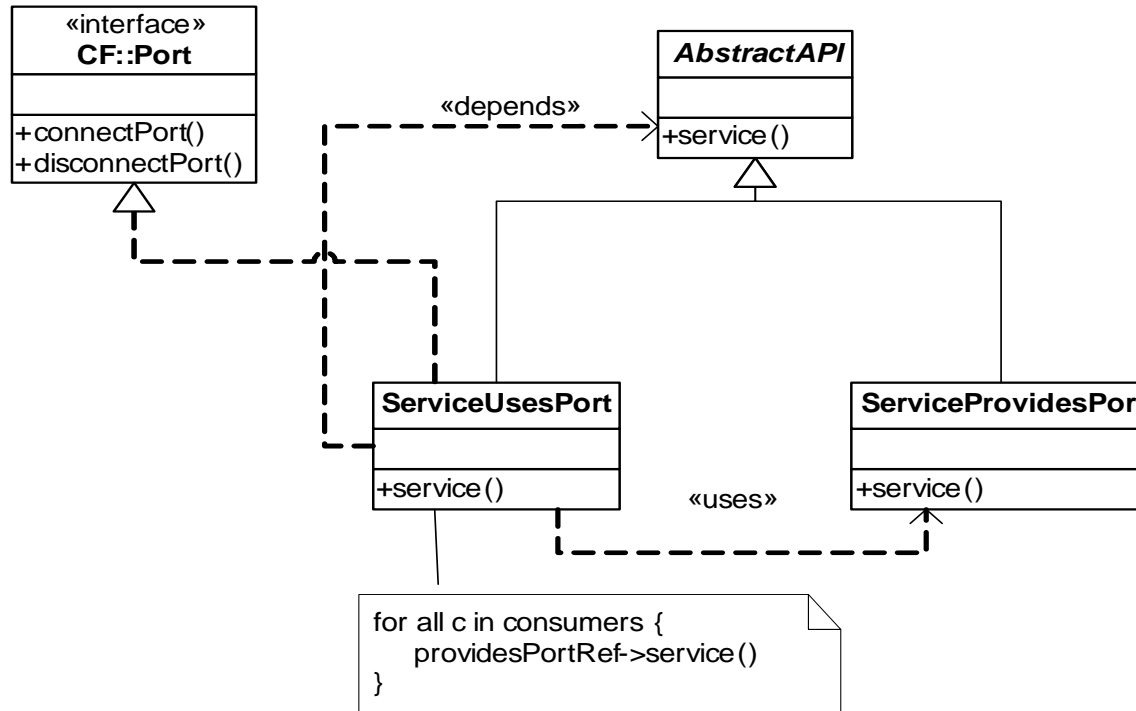
1. Gamma et. al. *Design Patterns*. Reading, MA: Addison-Wesley, 1995, p. 295
2. Ibid, p.293

Mapping the Observer Pattern to the SCA Domain



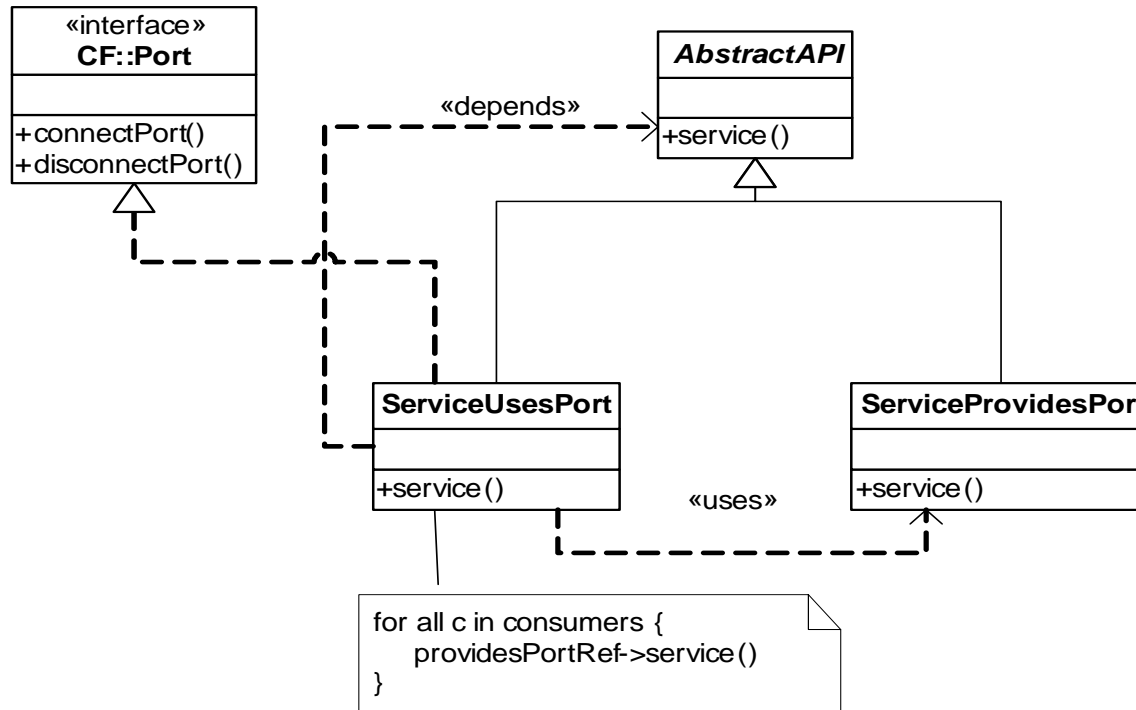
1. Gamma et. al. *Design Patterns*. Reading, MA: Addison-Wesley, 1995, p. 295

Mapping the Observer Pattern to the SCA Domain



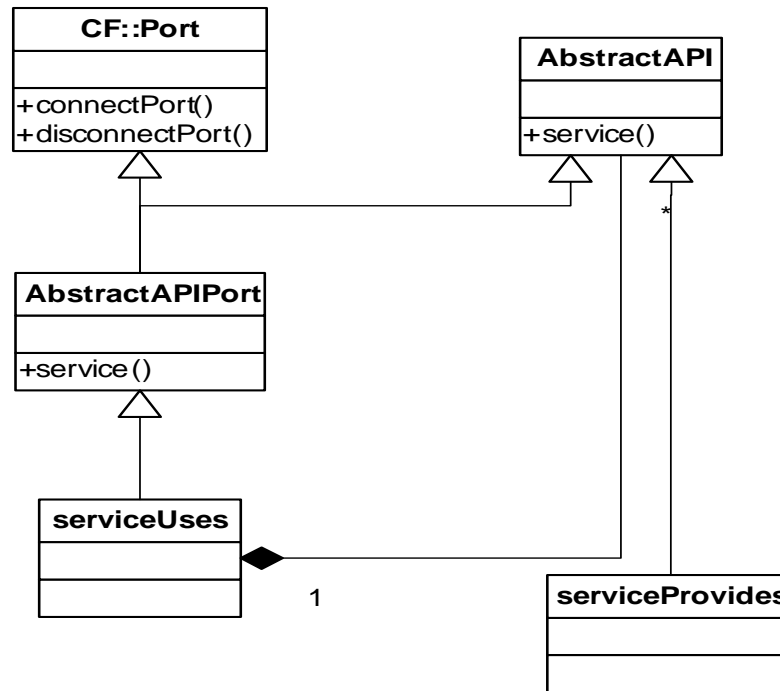
- **Segregate out the attach/detach observer functionality (connectPort/disconnectPort) into a separate interface since it is used by different clients than the clients of the AbstractAPI.**
- **Clients of CF::Port are typically the ApplicationFactory and the DomainManager. Clients of Abstract API are typically other components**

Mapping the Observer Pattern to the SCA Domain



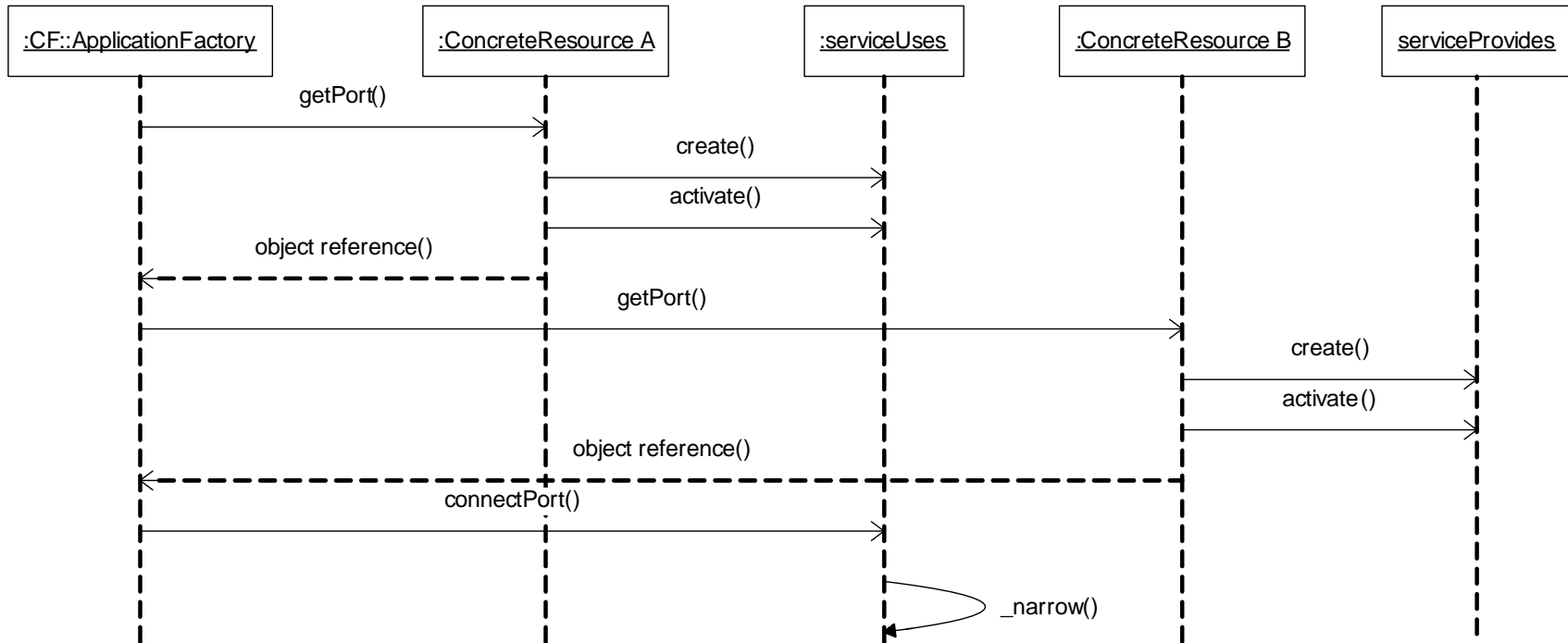
- **Collapse the Notify/Update interfaces into the Abstract API and use the Proxy pattern there.**
- **Allow usesports to be connected to a programmable number of provides port and have the usesport disseminate calls to all connected provides port.**

Uses and Provides Ports – The high level plumbing of the SCA – a Summary



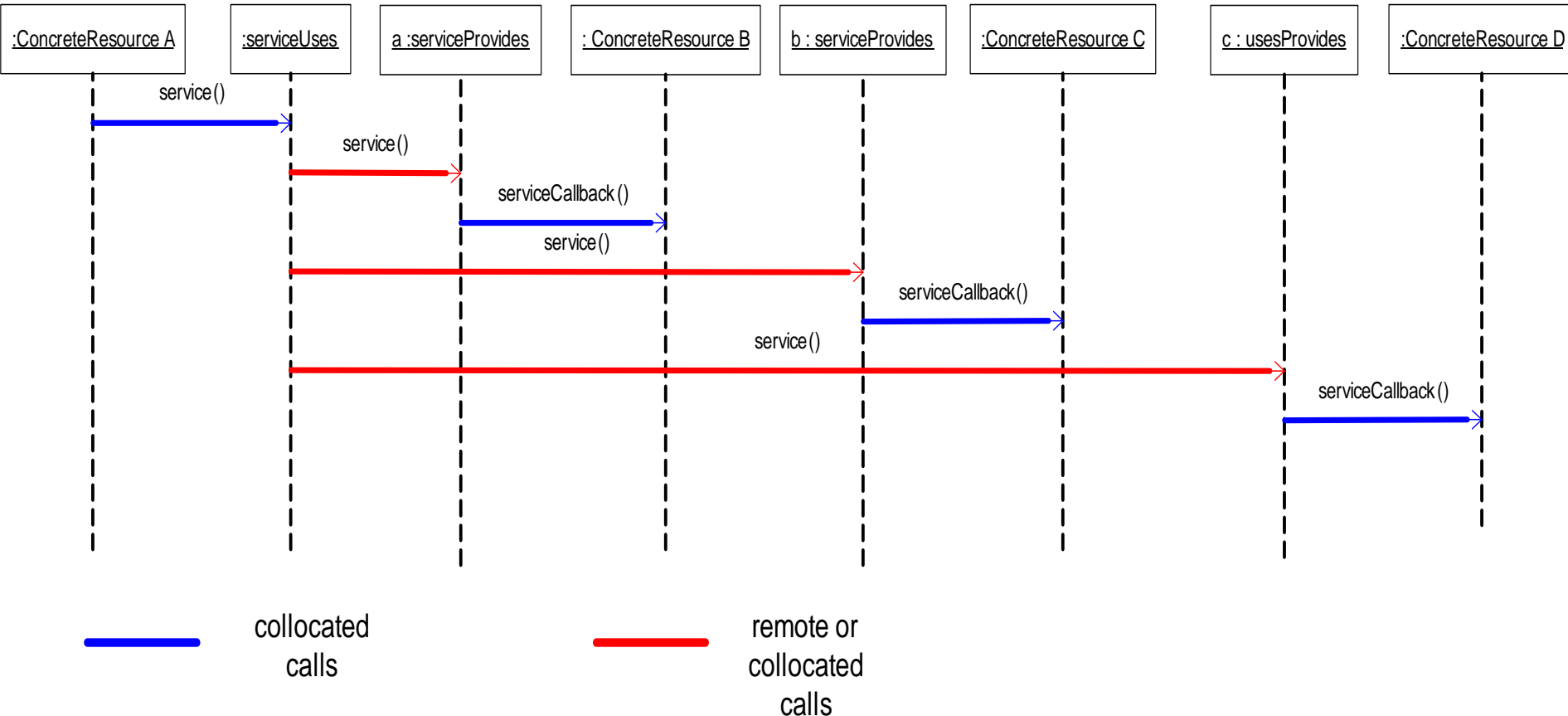
- Put connection related responsibilities inside the aggregated ports.
- Ports handle dissemination and receipt of port information.
- Uses side provides fan-out to provides side. This fan-out is transparent to the component. Notice one to many cardinality on serviceUses
- Uses side also handles distributed communication and acts as a proxy to the, possibly remote, provides port.
- Decouple inter component plumbing from the component proper

The Dynamics – Making the connection



- In the SCA Domain, the uses port performs the `_narrow` to the correct interface to ensure that provides ports are connected to the correct uses port and vice versa.
- To ensure uniqueness, the id passed both the `getPort()` and `connectPort()` methods are UUIDs defined in the XML files that describe the SCA Application

The Dynamics – Sending data between components



- In the SCA Domain, the uses port performs the `_narrow` to the correct interface to ensure that provides ports are connected to the correct uses port and vice versa.
- To ensure uniqueness, the id passed both the `getPort()` and `connectPort()` methods are UUIDs defined in the XML files that describe the SCA Application

Consequences of using Extension Interface/ Extension Object, Proxy and Observer Patterns together in the SCA

- On the uses side, the ConcreteResource code is simplified. Much like an the internals of an integrated circuit send signals out a single lead, the logic of the Resource is not complicated by fan-out semantics or the complications of making the remote call or collocated call to the consumers.
- On the provides side, due to the ability to have multiple consumers, the demuxing of incoming calls to specific functionality is achieved via simple callback registration and dispatch.
- Adding new interfaces is simplified. It boils down to simple registration of the port with the ConcreteResource for later return from the getPort call.

Consequences of using Extension Interface/ Extension Object, Proxy and Observer Patterns together in the SCA

- **The key abstraction, CF::Resource, does not polluted with operations that are specific to a specific client or waveform component. More flexible than extending the interface via subclassing.**
- **Dealing with extension interfaces can be more complicated from a clients perspective but this complication is cordoned off to the core interfaces of the Core Framework. These interfaces are the ApplicationFactory and the DomainManager. These are the interfaces that deal with the querying of components for extension interfaces (getPort) and “soldering” together the connections (connectPort and disconnectPort) and the necessary error/exception handling if the interface does not exist or the connection fails. The ports themselves can verify that they are being connected to the correct type of port (via the _narrow call). Component developers need not concern themselves with this complication**

Consequences of using Extension Interface/ Extension Object, Proxy and Observer Patterns together in the SCA

- Using the Extension Object Pattern, the SCA Application can export its different roles depending on what interfaces are requested and supplied by particular clients. For example, certain clients are interested in the SCA app from a data throughput standpoint versus built in test clients etc.
- Security – both the extension object and proxy patterns lend themselves well as mechanisms to support security mechanism. Request for interfaces can be validated and smart proxies can be used to control traversal from usesport to providesport.
- Allow for event sinks and sources and log sinks and sources to be easier added to Resources.

SCA Component Deployment

Component Deployment

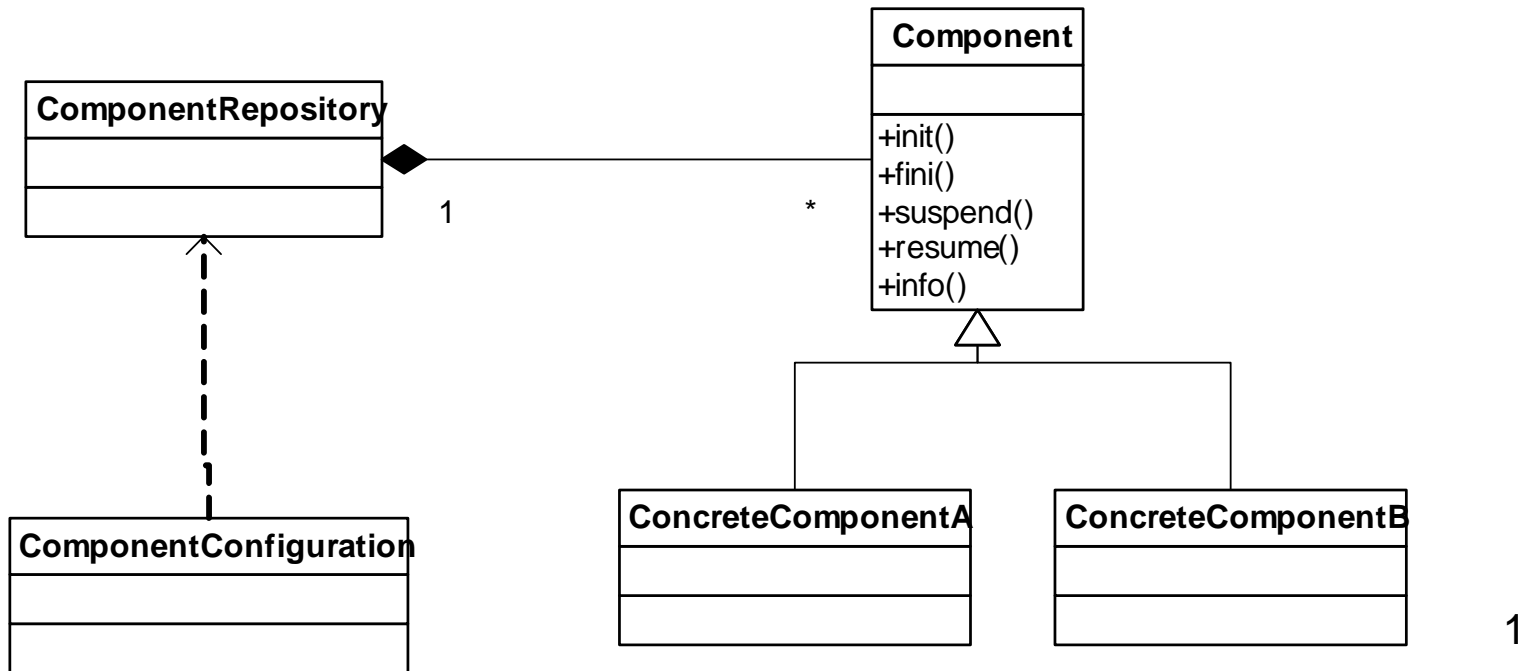
- **The question: What is the best way to deploy SCA components?**
- **The problems:**
 - **Due to the complexity of radio applications and the nature of software defined radios, developers often do not know early on how best to distribute functionality across system resources**
 - **This functionality may need to be distributed differently on different SDR platforms**
 - **Different SDR platforms may require different component implementations**

Component Deployment

- **The guiding principles:**
 - **The component developer should not need to commit a priori to a particular configuration of radio components**
 - **These configurations should be allowed to be committed to as late as possible, even possibly at run time**
 - **Configuration, Initialization, and deployment should be managed by a central mechanism**
- **The pattern: Component Configurator¹**
- **The solution:**
 - **Define a system that uses component and assembly descriptors**

1. Schmidt, et al. *Pattern-Oriented Software Architecture*, Volume 2, Wiley, 2000, p. 75

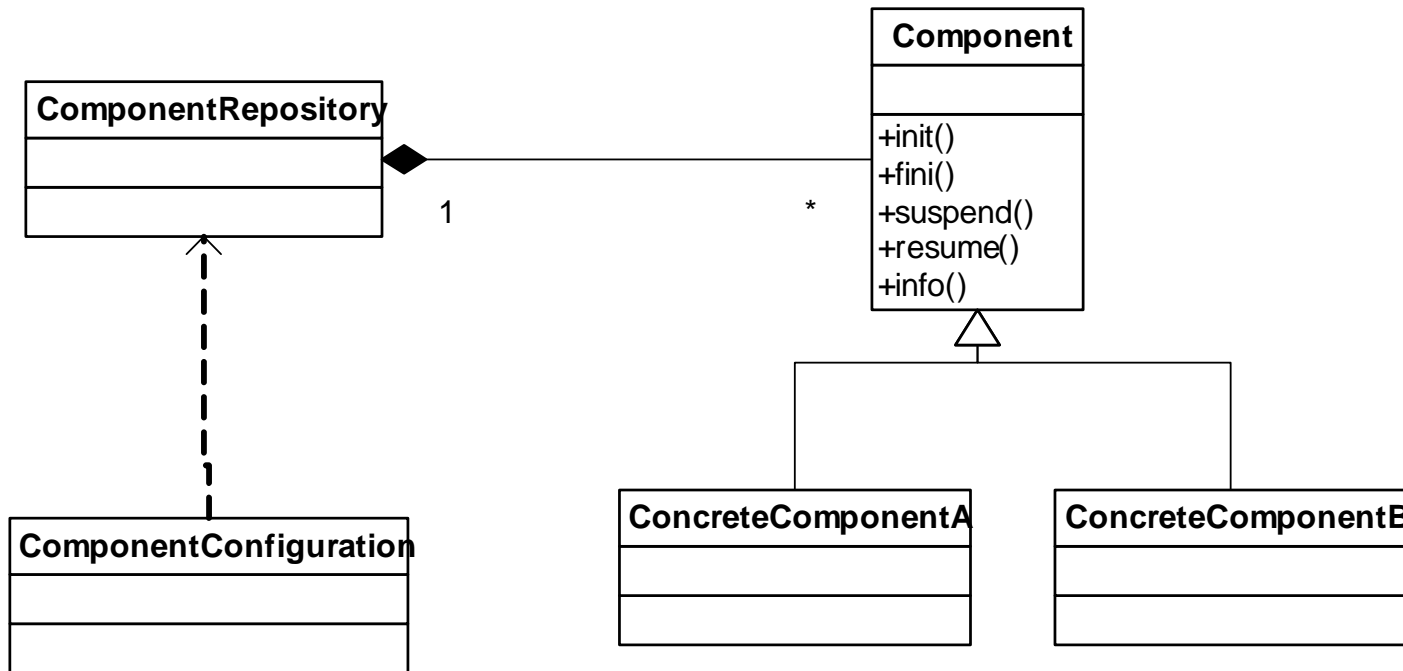
The Component Configurator Pattern - Structure



- **Component** – “defines a uniform interface that can be used to configure and control the type of application service or functionality provided by a component implementation”²

1. Schmidt, et al. *Pattern-Oriented Software Architecture*, Volume 2, Wiley, 2000, p. 80
 2. *Ibid*, p.79

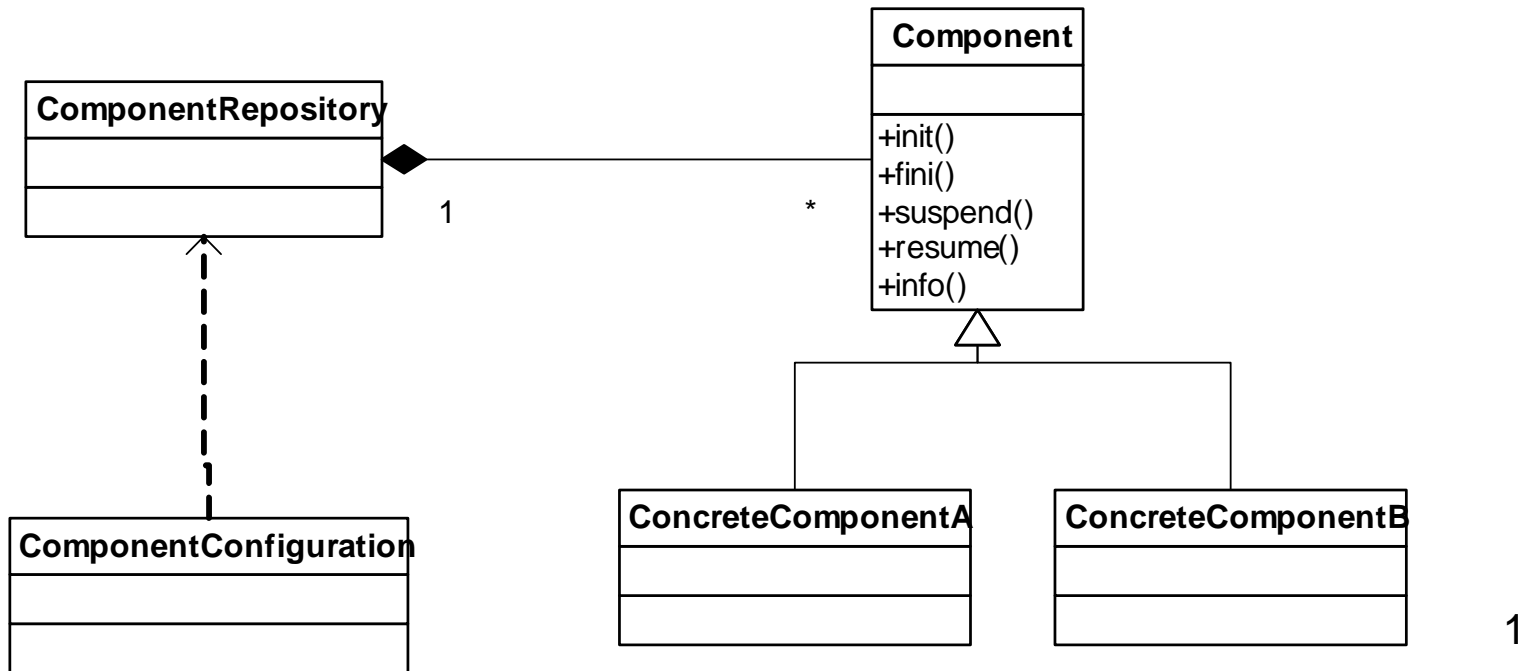
The Component Configurator Pattern - Structure



- **Component Repository – “manages all concrete components that are configured currently into an application. This repository allows system management applications or administrators to control the behavior of configured concrete components via a central administrative mechanism”²**

1. Schmidt, et al. *Pattern-Oriented Software Architecture*, Volume 2, Wiley, 2000, p. 80
 2. *Ibid*, p.79

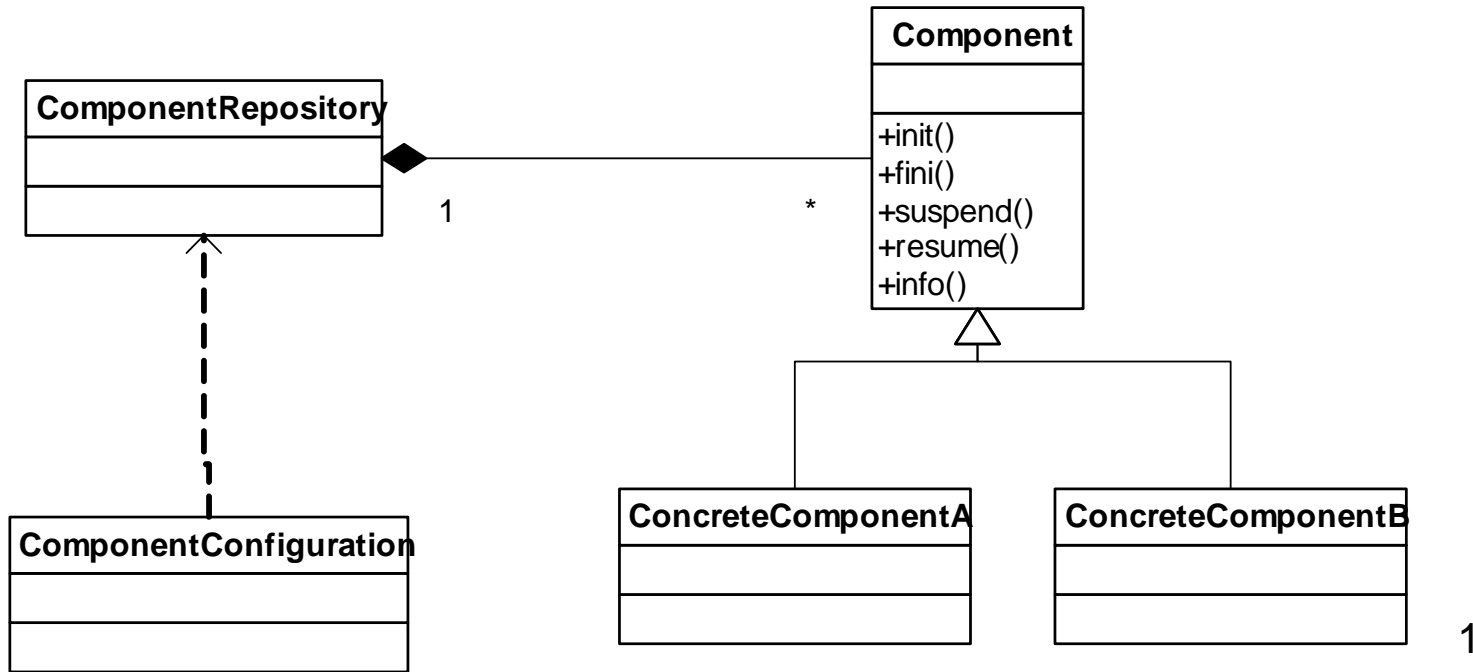
The Component Configurator Pattern - Structure



- **Concrete Component** – “Implement the component control interface to provide a specific type of component. Concrete Components are packaged in a form that can be dynamically linked and unlinked into or out of an application at run-time, such as a DLL”²

1. Schmidt, et al. *Pattern-Oriented Software Architecture*, Volume 2, Wiley, 2000, p. 80
 2. *ibid*, p.79

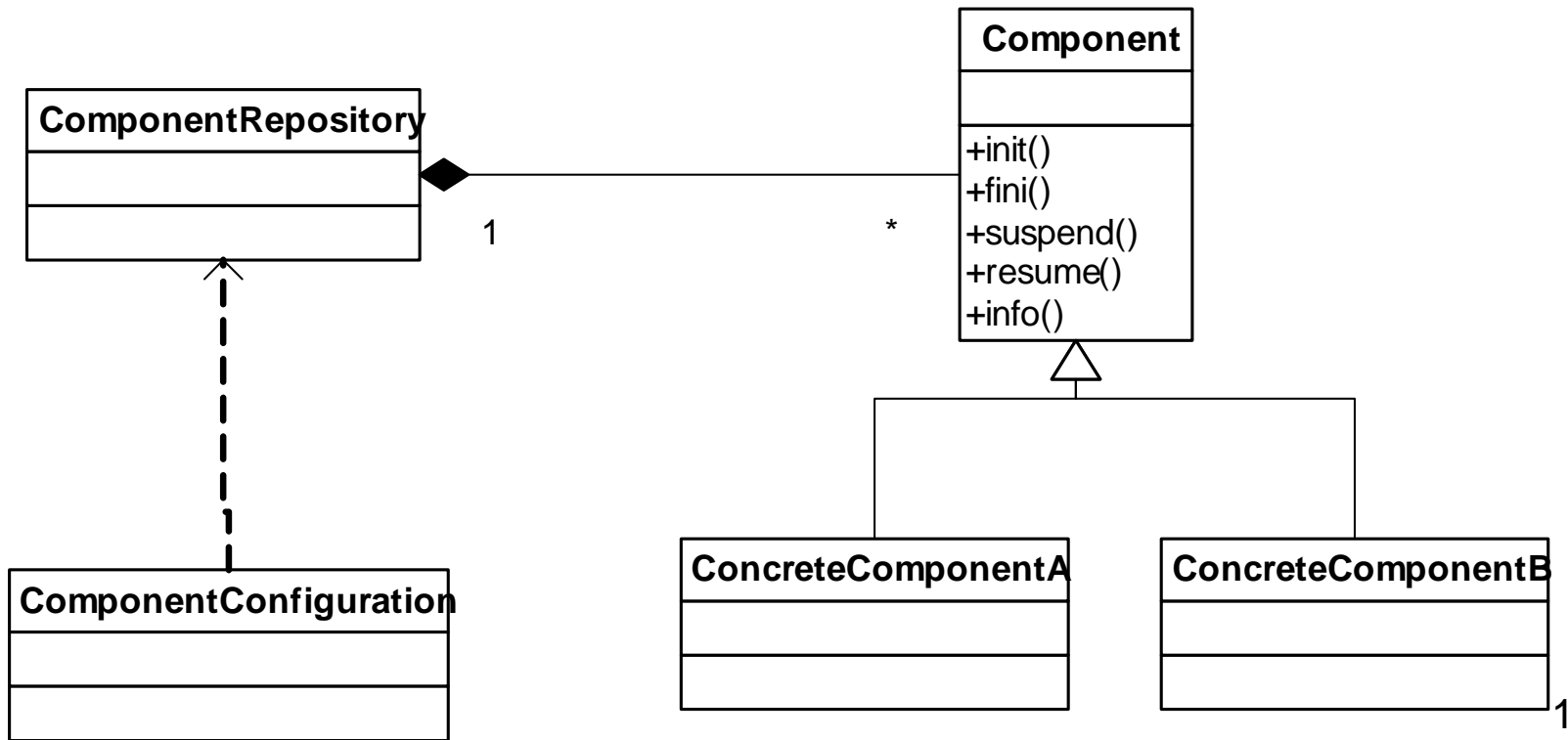
The Component Configurator Pattern - Structure



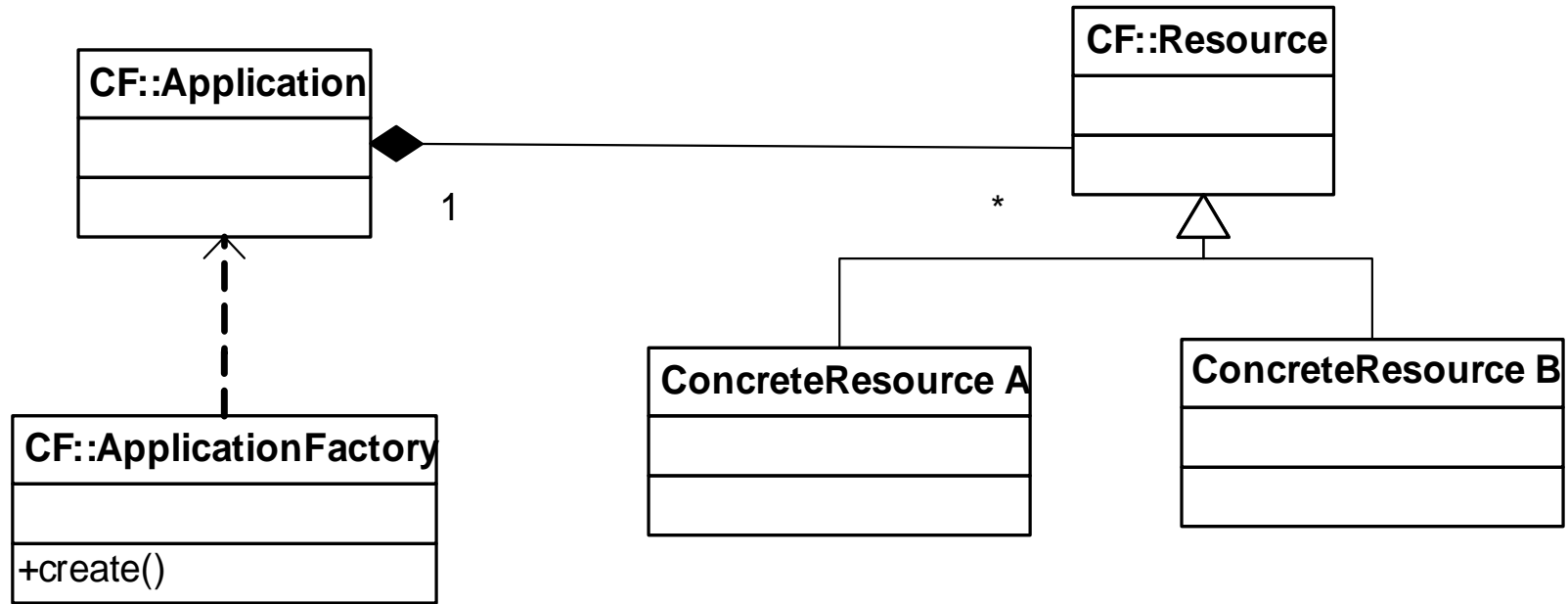
- **Component Configurator** – “uses the component repository to coordinate the (re)configuration of concrete components. It implements a mechanism that interprets and executes a script specifying which of the available concrete components to (re)configure into the application via dynamic linking and unlinking DLLS”²

1. Schmidt, et al. *Pattern-Oriented Software Architecture*, Volume 2, Wiley, 2000, p. 80
 2. Ibid, p.79

Mapping the Component Configurator Pattern to the SCA Domain

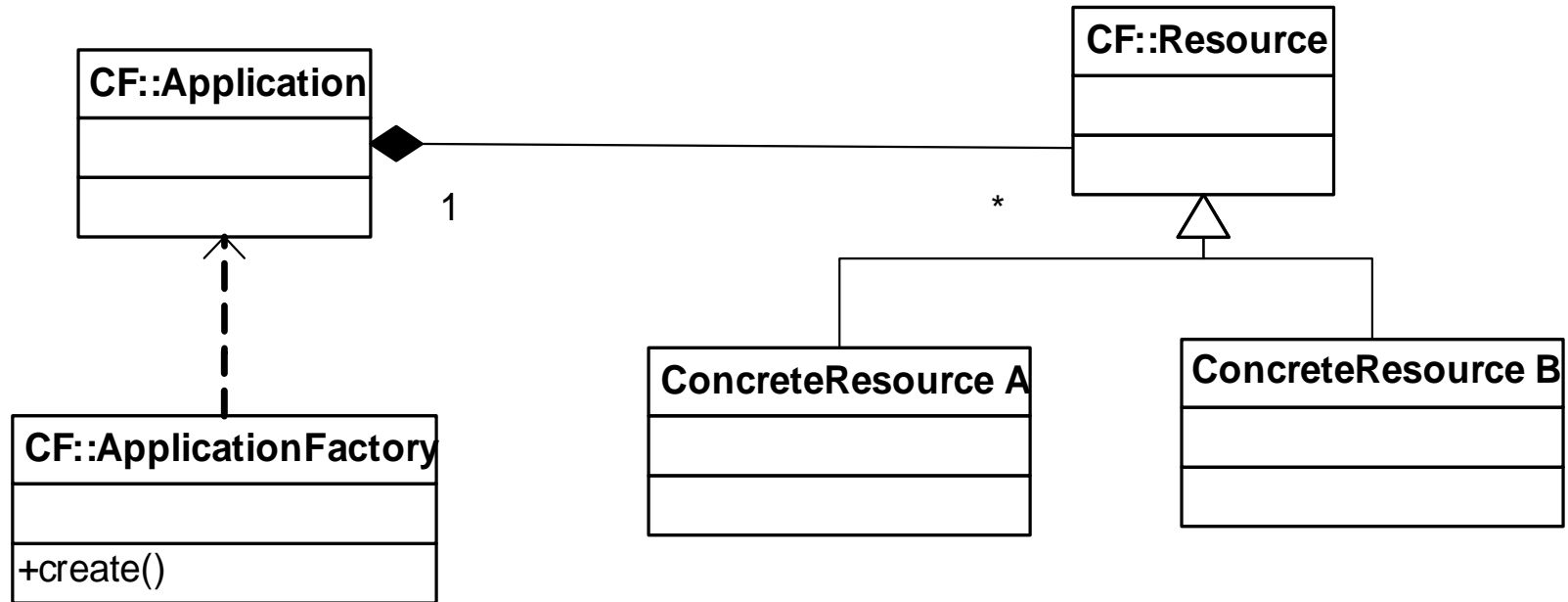


1. Schmidt, et al. *Pattern-Oriented Software Architecture*, Volume 2, Wiley, 2000, p. 80



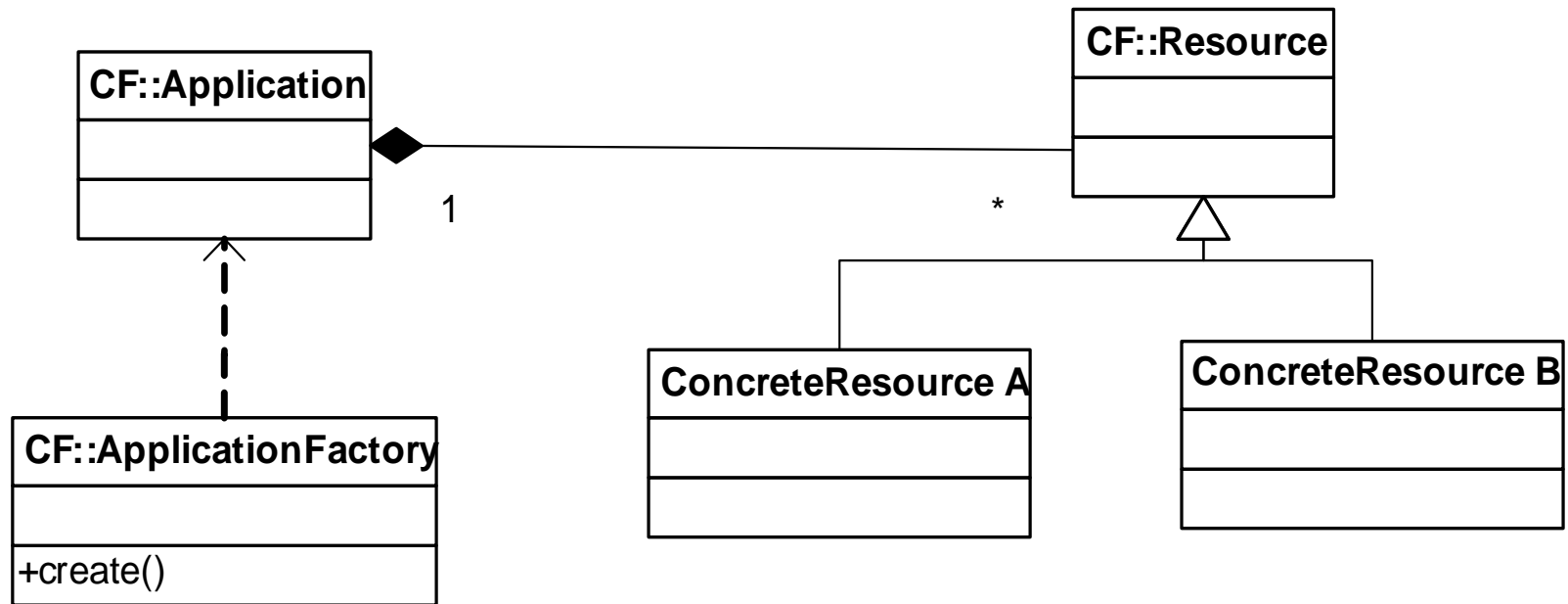
- **CF::ApplicationFactory** – This abstraction plays the role of the Component Configurator. The Domain Profile XML files play the role of the script interpreted by the ApplicationFactory

Mapping the Component Configurator Pattern to the SCA Domain



- **CF::Application** – plays the role of the Component Repository and is the aggregate abstraction that is presented to the outside world as a Façade for the entire waveform.

Mapping the Component Configurator Pattern to the SCA Domain



- **CF::Resource** – plays the role of **Component**
- **ConcreteResource** – plays the role of the **Concrete Component**

- **The Component Configurator pattern calls for “defining a language for specifying component configuration directives. These directives supply the component configurator with the information it needs to locate and initialize a components implementation at runtime.”¹**
- **In the SCA Domain, this script corresponds to the Domain Profile XML files and the corresponding parser used to read it. The information contained in these files directly tells the ApplicationFactory which implementations of which components to deploy and on which devices.**

1. Schmidt, et al. *Pattern-Oriented Software Architecture*, Volume 2, Wiley, 2000, p. 86

- **The Component Configurator Pattern calls for “Implementing the dynamic configuration mechanism that the configurator uses to link and unlink components”¹**
- **In the SCA, the Component Configurator implementation (ApplicationFactory) uses the CF::LoadableDevice and CF::ExecutableDevice for its explicit dynamic linking mechanism and API.**

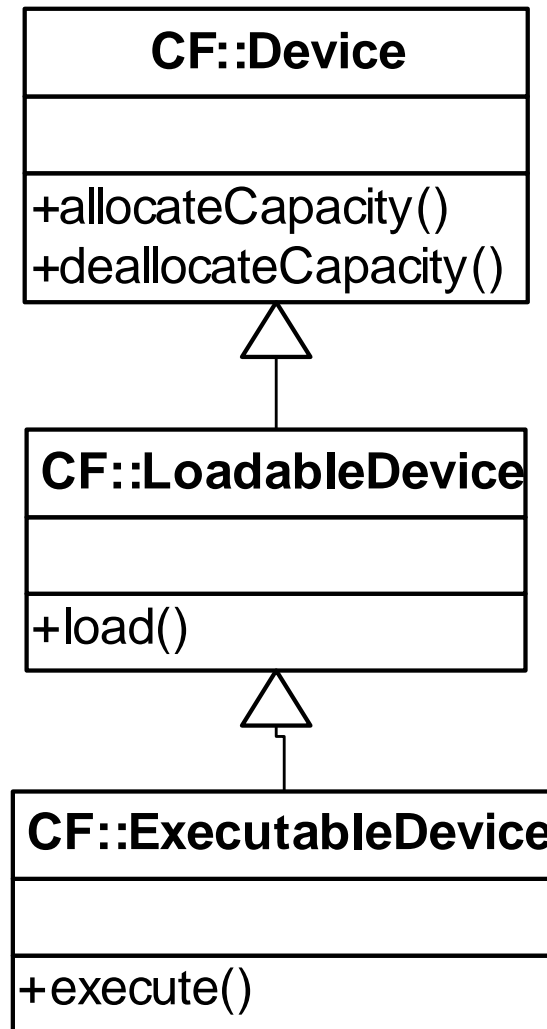
1. Schmidt, et al. *Pattern-Oriented Software Architecture*, Volume 2, Wiley, 2000, p. 88

Component Configurator – Benefits Summary

- **Waveform applications can load and unload their component implementations at run-time without having to modify or statically relink the entire application.**
- **Waveform applications can be easily redeployed and redistributed depending on changes in operation environment (e.g. load balancing, tighter security concerns)**
- **Deployment of waveforms no longer coupled to the deployment of the Core Framework on any other part of the Operating Environment.**
- **CF::Application provides a centralized repository and administrative mechanism for waveform management**
- **SAD file provides fast and easy mechanism to change the “schematic” of the waveform without recompiling or relinking.**

SCA Logical Device Design

The CF::Device hierarchy



The Design of CF::Device

- **The responsibilities of CF::Device, CF::LoadableDevice and CF::ExecutableDevice**
 - Abstract the physical hardware from the application software
 - Provide access to hardware capabilities from application software
 - Support the load and execution of application software onto physical hardware
 - Provide state management facilities for the radio platform
- **The clients of CF::Device**
 - Since a Device is a CF::Resource, the same clients apply
 - In addition CF::Device clients require
 - Mechanisms to deploy SCA components
 - Mechanisms to validate sufficient physical resources are available for deployment
 - Facilities to query and manage the state of the radio

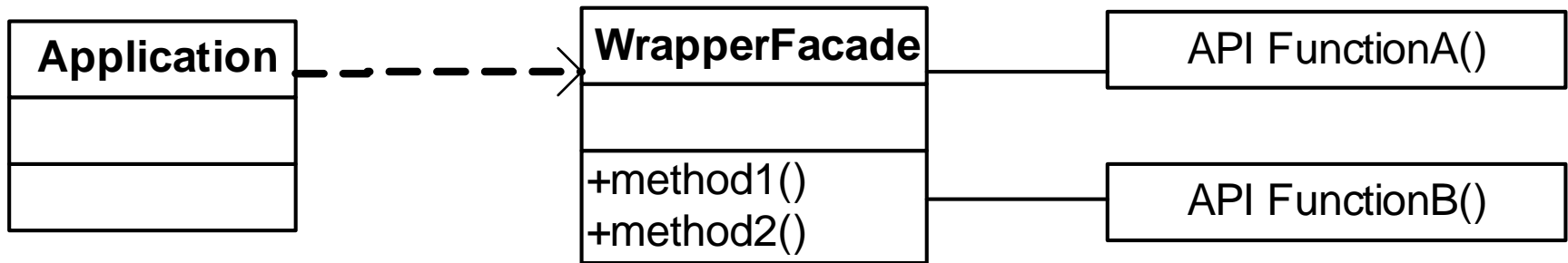
The Design of CF::Device

- **The guiding principles:**
 - **Decouple the allocateCapacity and deallocateCapacity mechanisms from the algorithms used to do the actual allocation and deallocation**
 - **Create simple cohesive and robust interfaces for complicated subsystems. Provide encapsulations of the functions and data provided by existing non-object-oriented APIs.**
 - **Abstract away the specific dynamic loading mechanism implementation of the physical hardware and decouple this from the parsing and processing of the configuration directives and descriptor files.**
- **The patterns: Wrapper Façade¹, Strategy²**

1. Schmidt, et al. *Pattern-Oriented Software Architecture*, Volume 2, Wiley, 2000, p. 47

2. Gamma et. al. *Design Patterns*. Reading, MA: Addison-Wesley, 1995, p. 315

The Wrapper Facade Pattern - Structure



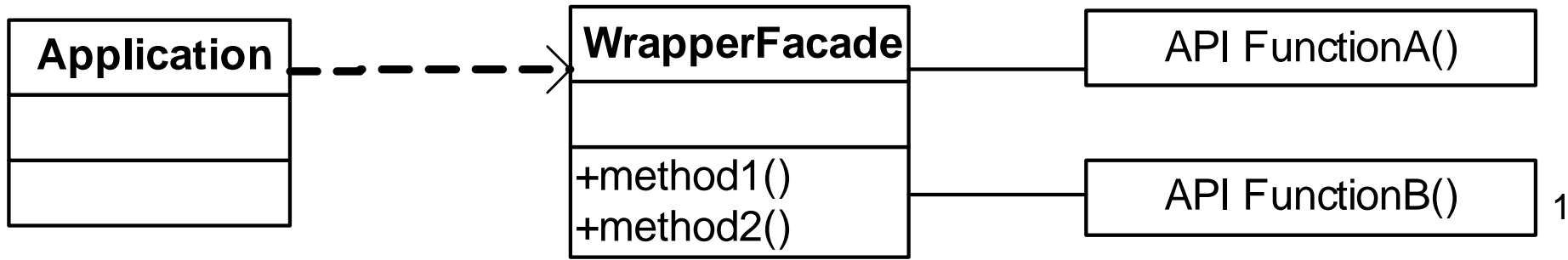
1

- **Application** – “The application code invokes a method on an instance of the wrapper façade”
- **Wrapper Façade** – “The wrapper façade forwards the request and its parameters to one or more of the lower-level API functions that it encapsulates, passing along any internal data needed by the underlying functions(s)”

1. Schmidt, et al. *Pattern-Oriented Software Architecture*, Volume 2, Wiley, 2000, p. 52

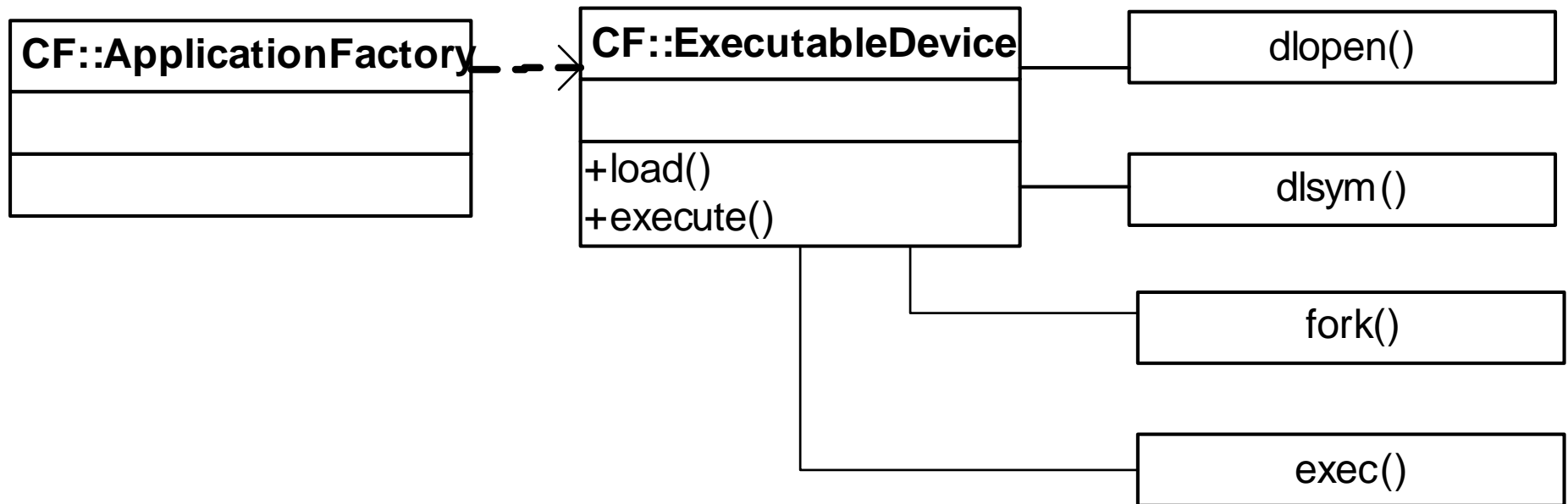
2. *Ibid.* p.53

The Wrapper Facade Pattern - Structure

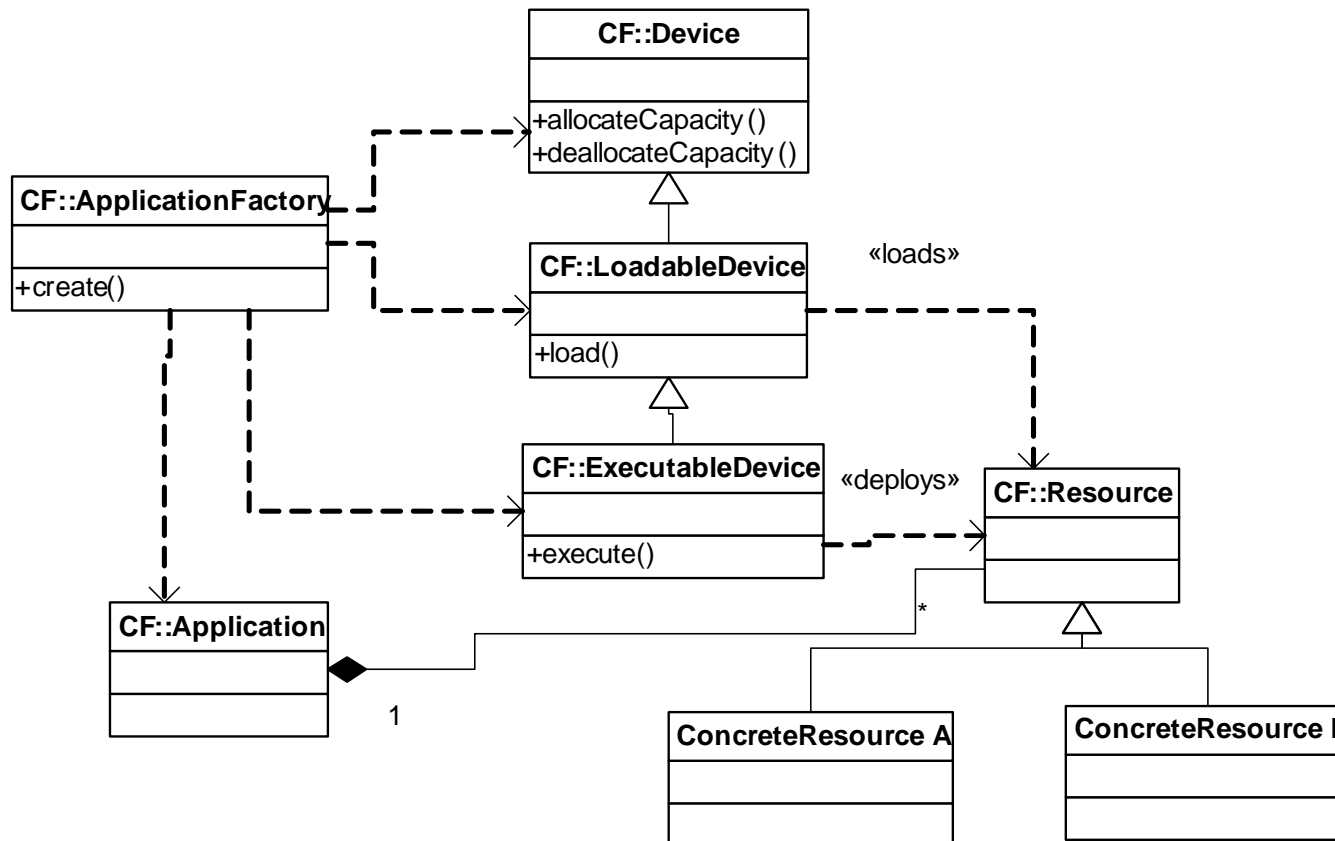


1. Schmidt, et al. *Pattern-Oriented Software Architecture*, Volume 2, Wiley, 2000, p. 52

The Wrapper Facade Pattern - Structure



Tying it back to the component configurator

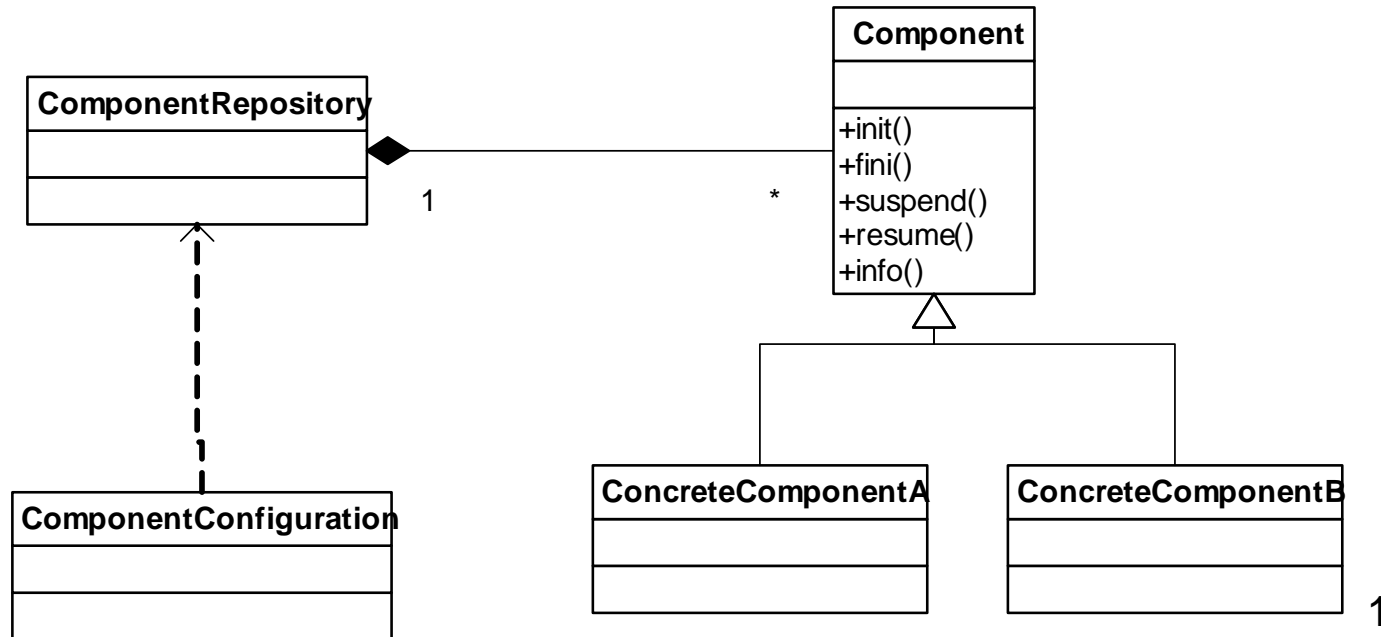


- **Loadable and Executable Devices provide the facades to the dynamic loading and linking mechanisms used by ApplicationFactory to deploy Resources and to the Application for teardown.**

The component configurator pattern, the CF::DeviceManager, CF::Devices and Services

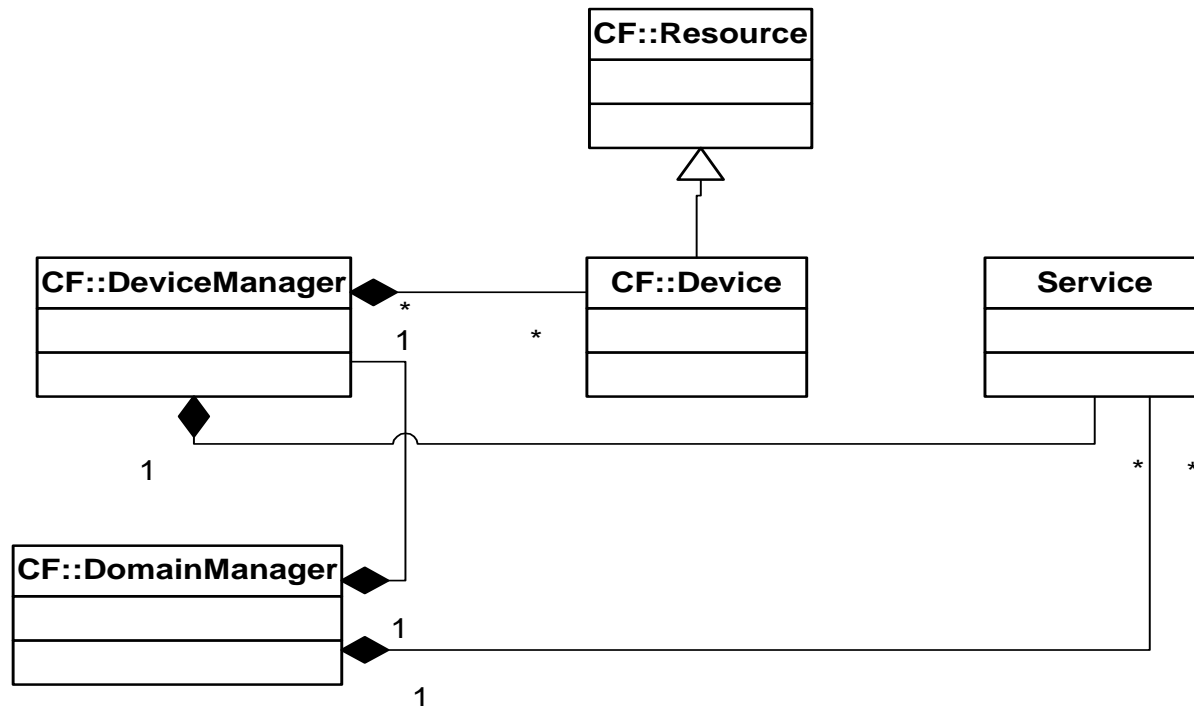
- There are two main areas in the SCA where the component configurator comes into play
 - Spawning and configuration of CF::Resources (waveform components)
 - Spawning and configuration of CF::Devices and Services
- As covered early, the CF::Resources fall into the domain of and are usually developed by Waveform vendors.
- CF::Resources and Services are usually developed by radio and radio platform vendors.

The component configurator pattern, the CF::DeviceManager, CF::Devices and Services

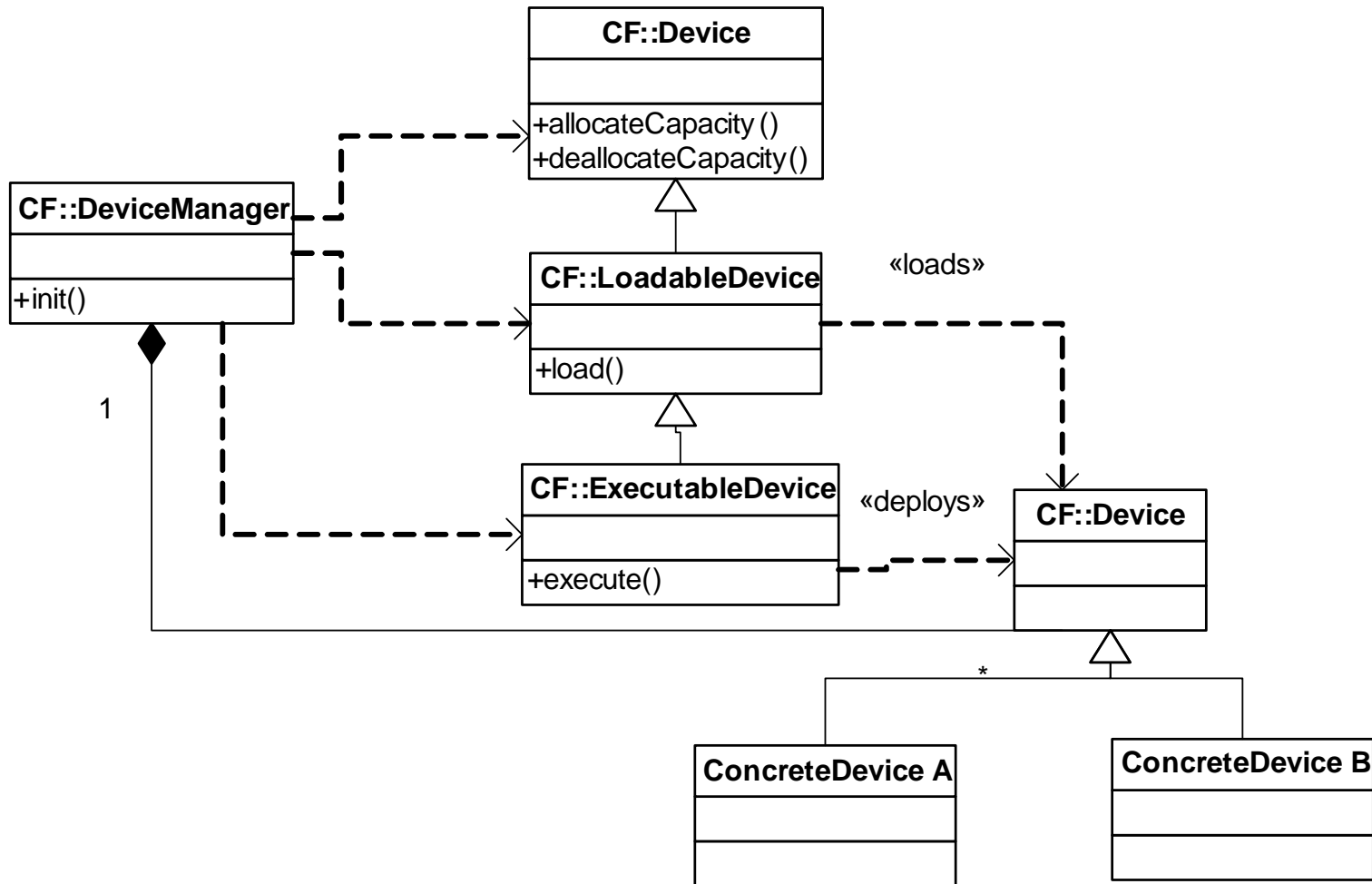


1. Schmidt, et al. *Pattern-Oriented Software Architecture*, Volume 2, Wiley, 2000, p. 75

The component configurator pattern, the CF::DeviceManager, CF::Devices and Services



The component configurator pattern, the CF::DeviceManager, CF::Devices and Services



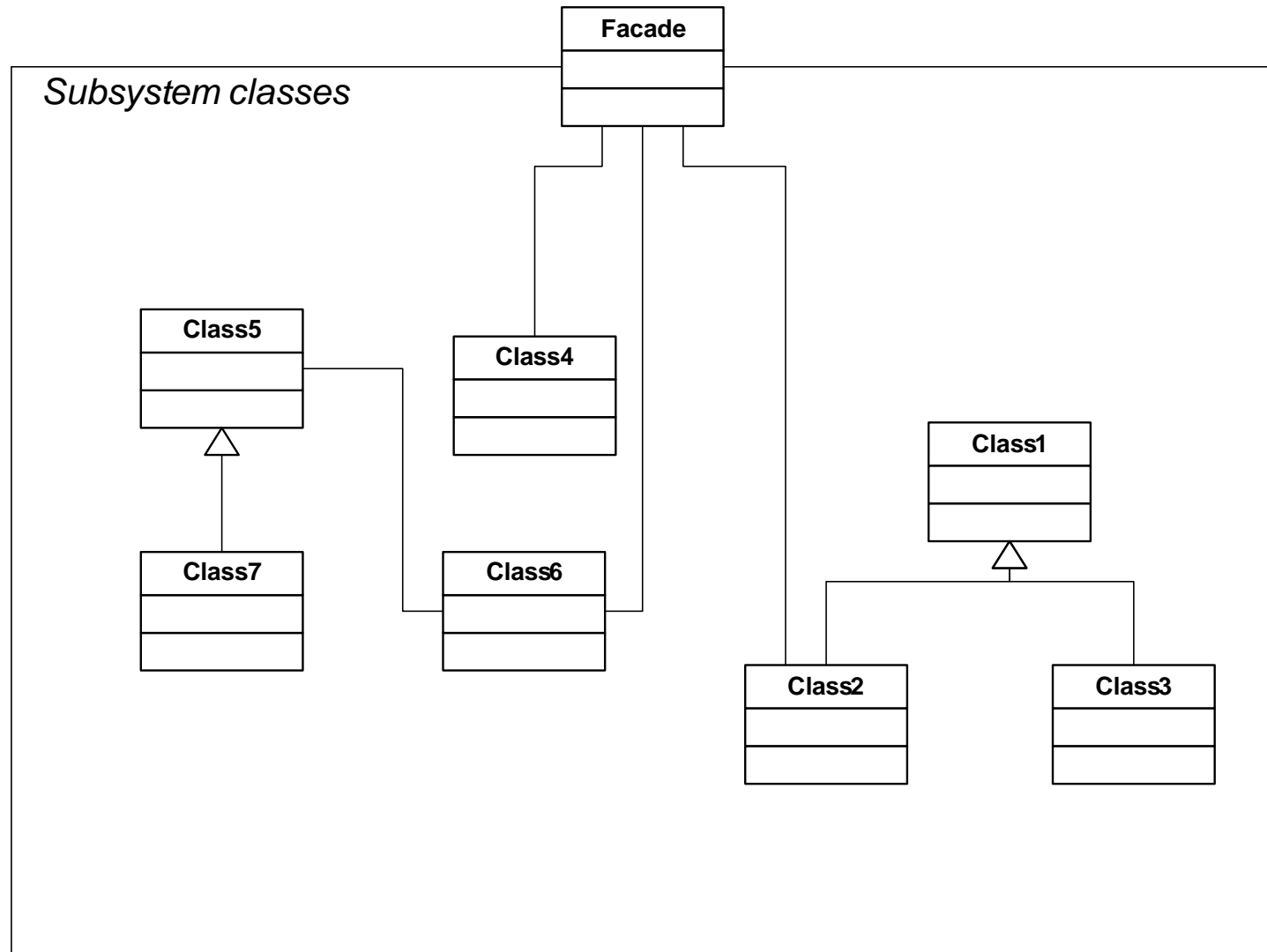
The component configurator pattern, the CF::DeviceManager, CF::Devices and Services

- With regard to the radio platform itself, the CF::DeviceManager play multiple roles in the component configurator pattern.
 - The component configurator itself
 - It can play the role of the handling the dynamic configuration mechanism
 - The component repository
- There is a separate XML file (DCD file) plays the role of the configuration script for the radio platform and is parsed by the CF::DeviceManager
- The CF::Devices and Service play the role of the component

Wrapper Façade Pattern – Benefits Summary

- **Simplify access to OS APIs (e.g. APIs for Threading)**

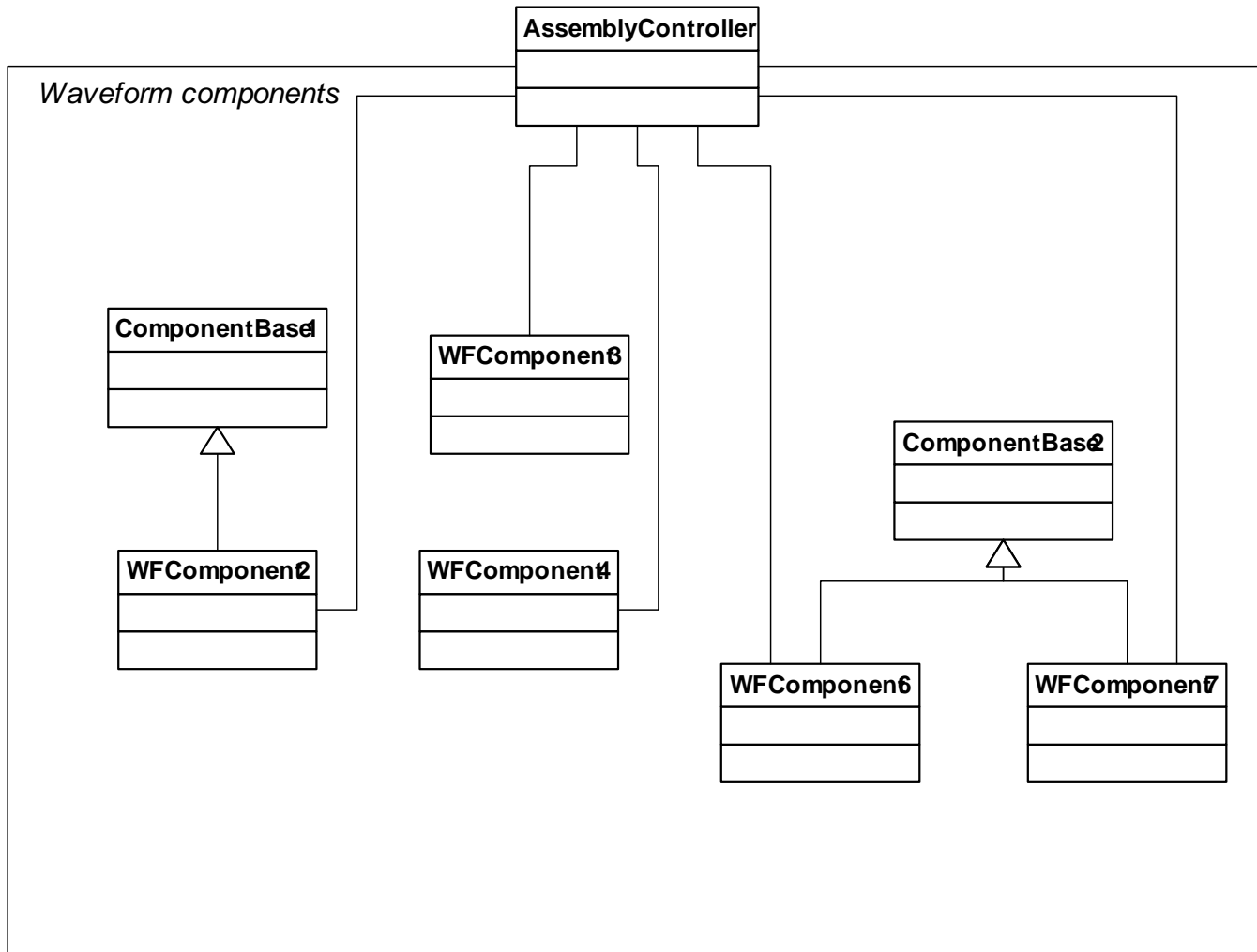
Facade Pattern



1

1. Gamma et. al. *Design Patterns*. Reading, MA: Addison-Wesley, 1995, p. 187

Facade Pattern continued



Façade Pattern - Participants

- **Façade**
 - “Knows which subsystem classes are responsible for a request”
 - “Delegates client requests to appropriate subsystem objects”¹
- **Subsystem classes**
 - “implements subsystem functionality”
 - “handle work assigned by the Façade object”
 - “have no knowledge of the façade; that is, they keep no references to it”²

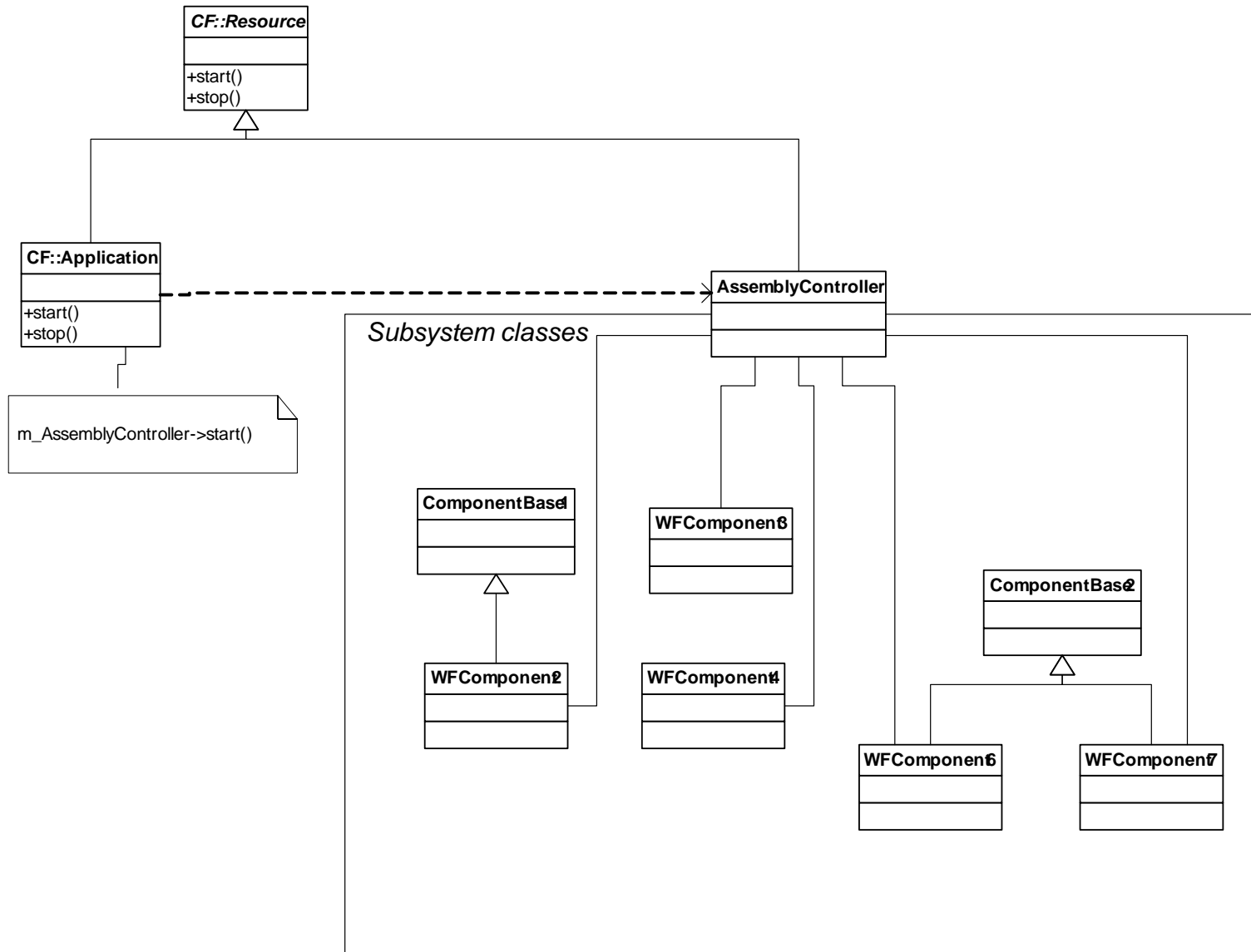
1. Gamma et. al. *Design Patterns*. Reading, MA: Addison-Wesley, 1995, p. 187

2. Ibid, p.187

Façade Pattern mapped to SCA Domain

- **The Assembly Controller functions as the Facade**
 - It has waveform specific knowledge of which components should receive configurations and in what order. It also can control the order of start up of the waveform components
 - It delegates commands to the waveform components
- **The waveform components function as subsystem classes**
 - implement waveform functionality
 - execute commands issued by the assembly controller
 - Do not maintain references to the assembly controller

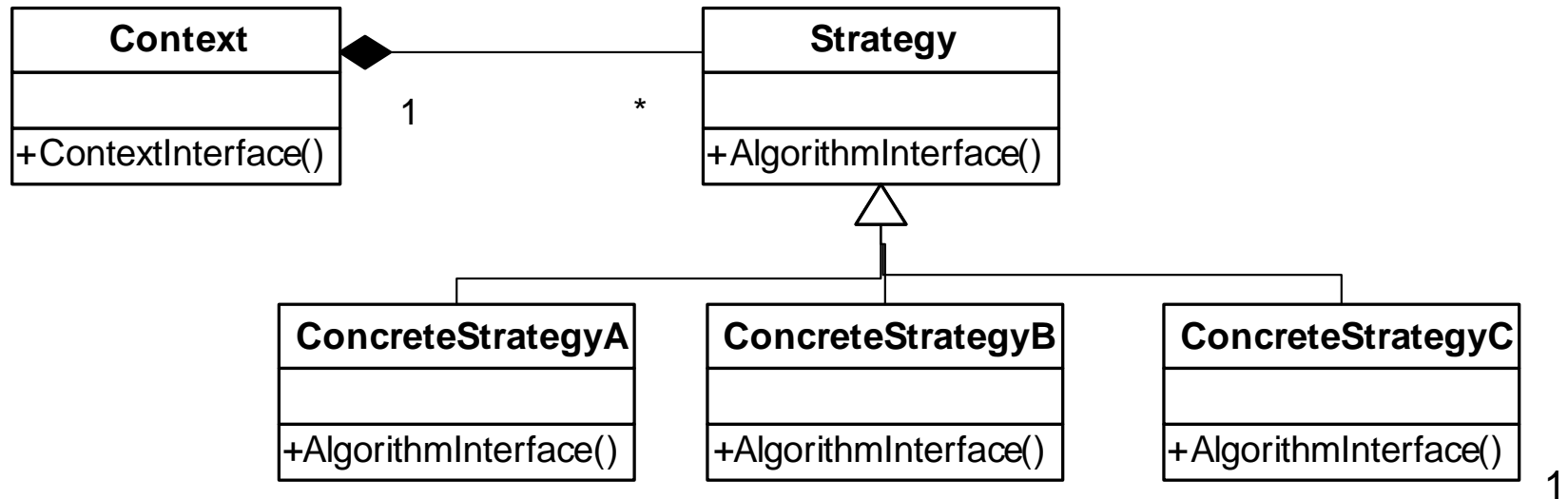
CF::Application and Assembly Controller – Proxy with Facade



Proxy and Facade Pattern – Benefits Summary

- **Complex waveforms hidden behind single abstraction**
- **Well known interface established between Core Framework and the Waveform (CF::Application and Assembly Controller bridge the two)**
- **Hide distribution mechanics (e.g. Components may be in separate process from each other or in the same address space)**
- **Seamlessly allow the addition of new connection semantics (e.g. fan-out and controlled number of connections)**

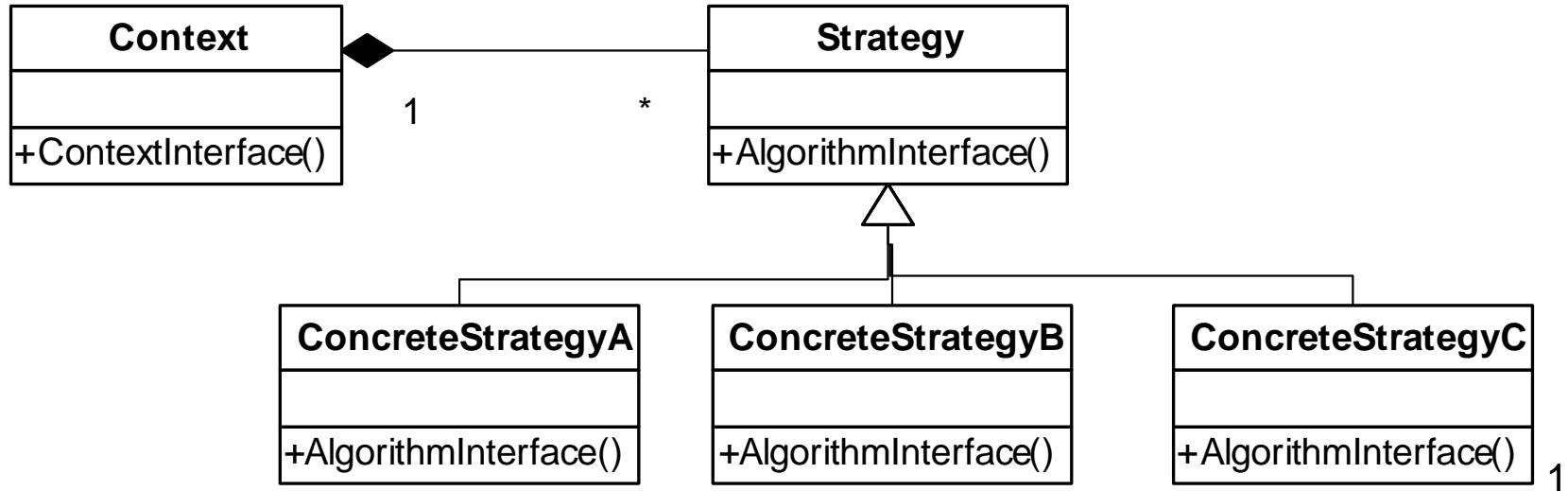
The Strategy Pattern - Structure



- “Strategy – declares an interface common to all supported algorithms
- ConcreteStrategy – implements the algorithm using the Strategy interface
- Context – configured with a ConcreteStrategy object”²

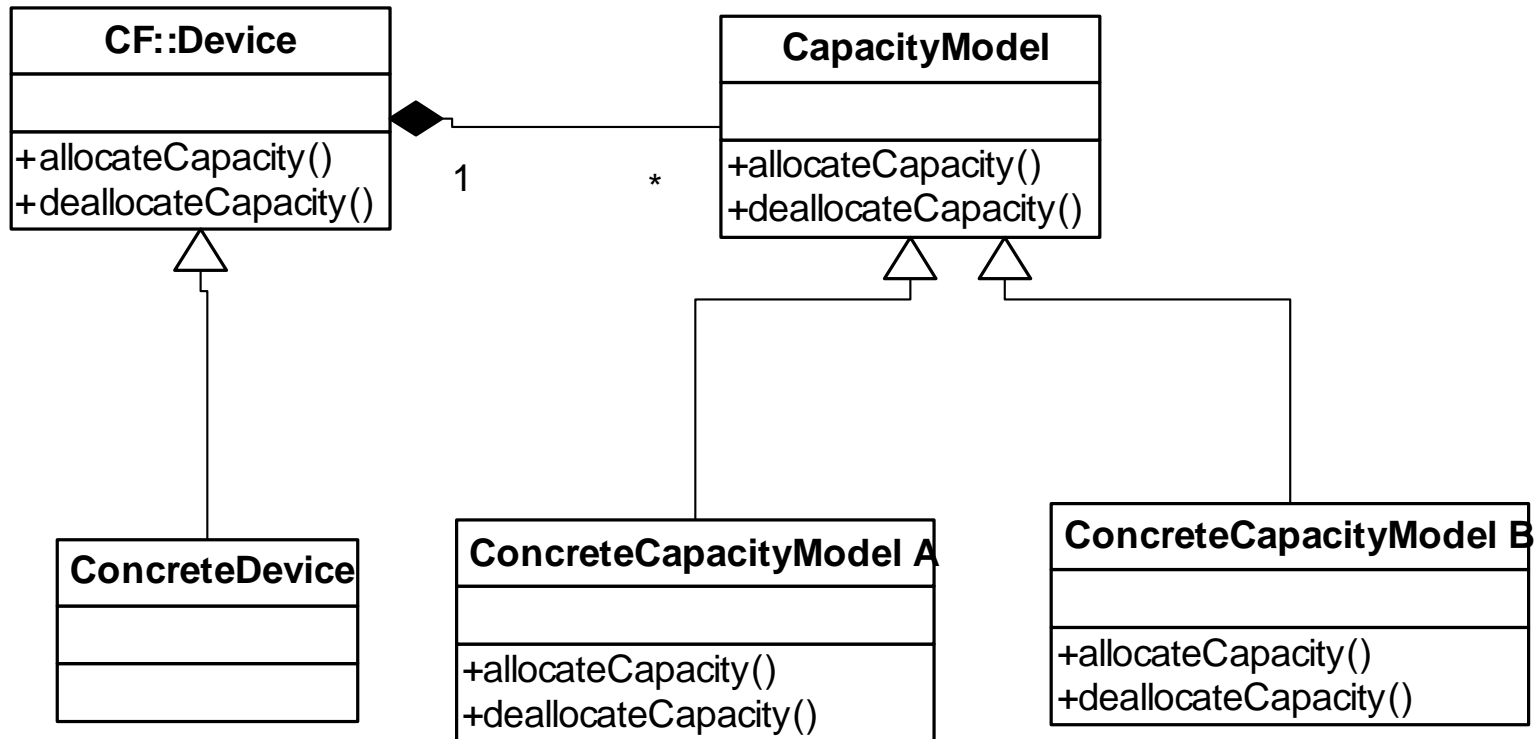
1. Gamma et. al. *Design Patterns*. Reading, MA: Addison-Wesley, 1995, p. 316
 2. Ibid p.210

Mapping the Strategy Pattern to the SCA Domain



1. Gamma et. al. *Design Patterns*. Reading, MA: Addison-Wesley, 1995, p. 316

Mapping the Strategy Pattern to the SCA Domain



- **CF::Device** and **Concrete Device** play the context role
- **CapacityModel** and **ConcreteCapacityModelA** play the Strategy and ConcreteStrategy role respectively
- This design allows the Capacity models to vary independently from the clients that use it (e.g. **CF::Device**)

Strategy Pattern – Benefits Summary

- **Decouples the Capacity Allocation mechanisms from the models that describe the capacity of the Device abstraction.**

Techniques to increase maintainability and correctness of components

- **The question: What refactorings can be applied to component development to increase maintainability and correctness?**
- **The problems:**
 - Much of the effort of the component developer/user is duplicated from component to component.
 - Much of the code is in support of SCA infrastructure.
 - The developers need to have an intimate understanding of the Resource and Device behaviors.
 - Bottom line: Resource and Device developer's are saddled with dealing with more detail than they need to.

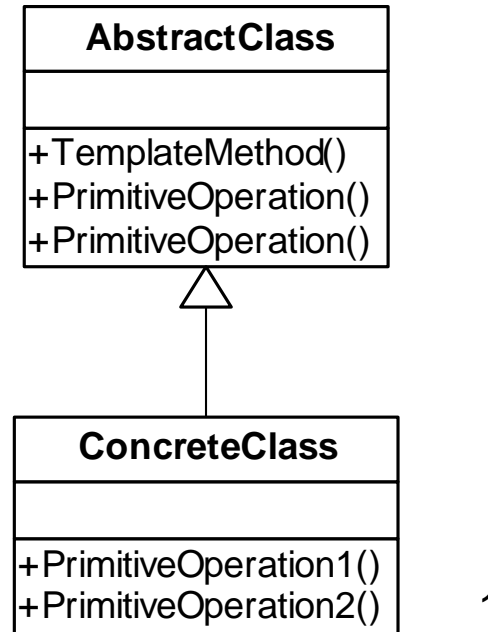
- **Guiding Design Principles**
 - **Dependency Inversion Principle¹**
 - **Single Responsibility Principle²**
 - **Eliminate repetitive/mechanical code constructs**
 - **Decouple the SCA mechanics from the implementation behavior of the components (business logic)**

- **The Solution:** Move common SCA behavior to a common base class and allow subclasses to re-define certain aspects of the implementation.

- **The Pattern: Template Method³**

1. Martin et. all, Agile Software Development, Prentice Hall 2003, p.127
2. Martin et. all, Agile Software Development, Prentice Hall 2003, p.95
3. Gamma et. al. *Design Patterns*. Reading, MA: Addison-Wesley, 1995, p. 325

The Template Method Pattern - Structure



- **“AbstractClass**

- Defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm
- Implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in **AbstractClass** or those of other objects.

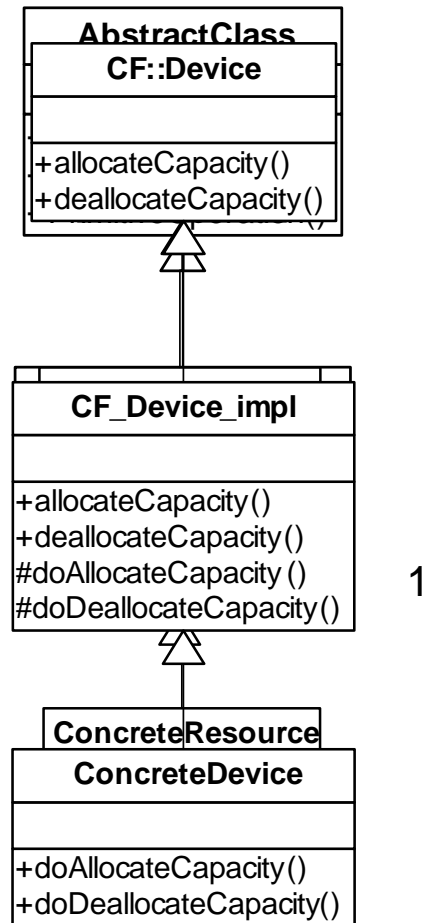
- **ConcreteClass**

- Implements the primitive operations to carry out subclass-specific steps of the algorithm”²

1. Gamma et. al. *Design Patterns*. Reading, MA: Addison-Wesley, 1995, p. 327

2. Ibid, p.327

Mapping the Template Method Pattern to the SCA Domain



1

1. Gamma et. al. *Design Patterns*. Reading, MA: Addison-Wesley, 1995, p. 327

Mapping the Template Method to the SCA – the specifics

- Can be used with both CF::Resource and CF::Device functionality as well as the base interfaces CF::PortSupplier, CF::PropertySet, etc.
 - e.g start, stop, configure , query, runTest, initialize, releaseObject, allocateCapacity etc.
- Each is implemented in the provided base class and defers any concrete component-specific behavior to the derived class through the use of non-public virtual functions associated with the interfaces.
 - e.g. doStart, doStop, doConfigure, doQuery etc.

Mapping the Template Method to the SCA – the specifics

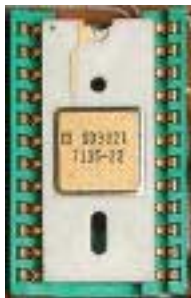
- Benefits
 - State management and the corresponding exception code for each operation can be moved to the new intermediate base classes while deferring concrete class-specific functionality to the concrete classes.
 - Allows for default behavior to exist in the new base classes.
 - e.g. start() throwing an exception if the component is not in the initialized state.
 - Connections management functionality can be dealt with in the provided base classes.
- Containers can be provided in the base classes to hold the necessary data for the concrete resources
 - e.g. collections of ports, collections of properties

Why do all this?

The Abstractions

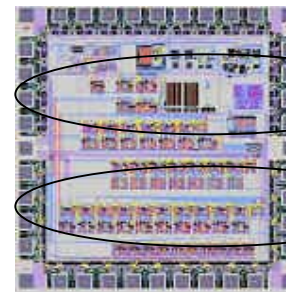
- There are two parts involved in developing an SCA component.
 - The SCA related part – the part that makes it compliant to the specification
 - Allowing the core framework to use it
 - The business logic part – the part that implements the functionality the component offers.
- Analogous to integrated circuits

How the outside world sees the component



It adheres to a framework allowing it to work with other components, e.g. - 5 volt logic

Internally things look quite a bit different



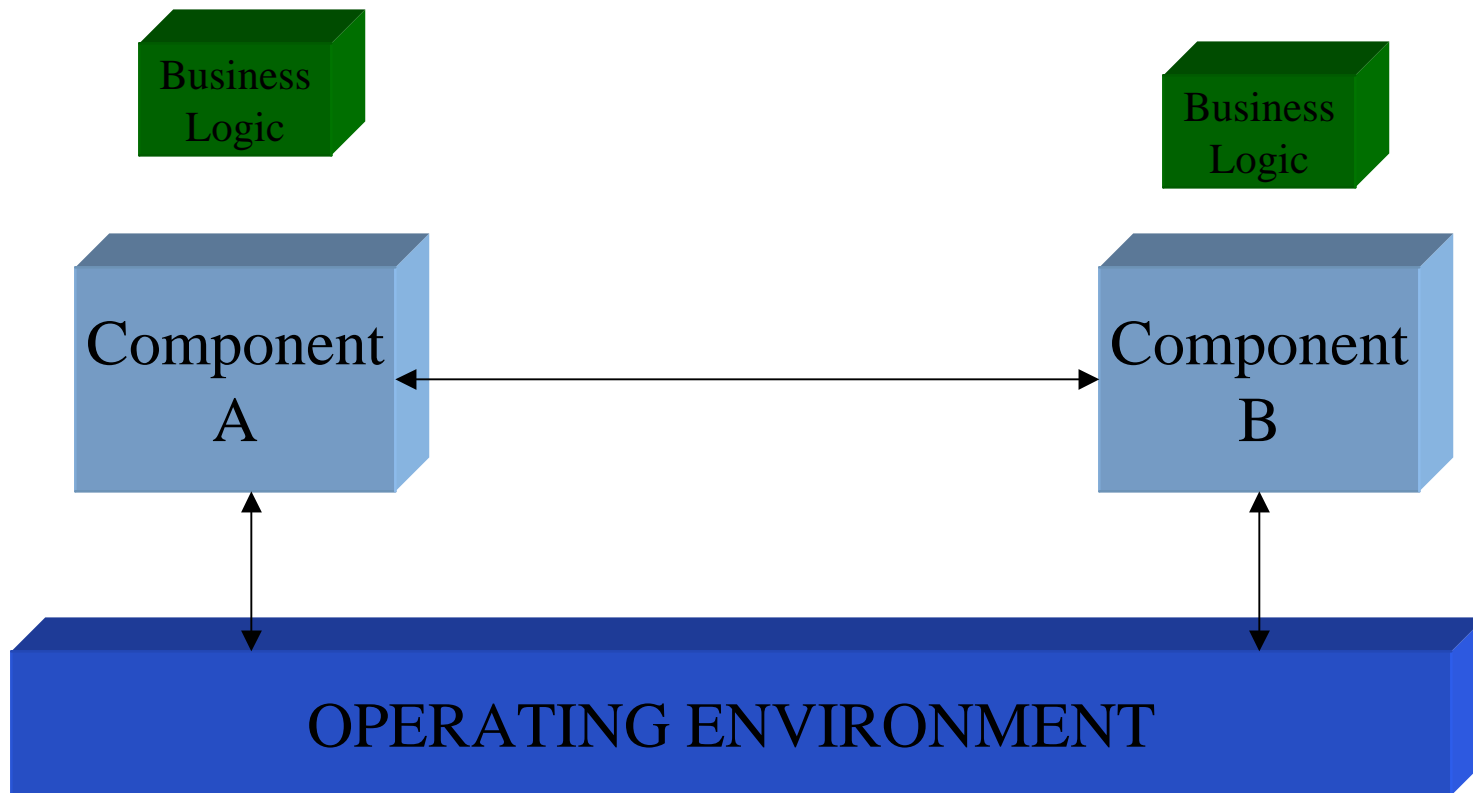
Waveform specific code

Common SCA component code – allowing it to work with framework

- **Waveform Development (CF::Resource)**
 - **Business logic parts must be developed by domain experts and are quite specialized**
 - **The parts that interface to the SCA core framework and other components are quite re-usable**
 - **Need to capture the recurring patterns/code**
 - **Implementations of these patterns can be supplied to developers**
- **Radio Platform Development (CF::Device)**
 - **Just as with waveform component development, the responsibility boundaries are very definite**
 - **SCA related radio platform component implementations can also be separated from the business logic**
 - **Porting of Devices to new platforms is eased due to defined separation of responsibilities**
 - **Frameworks to support radio hardware allocation capacity models can be provided to ease Device development**

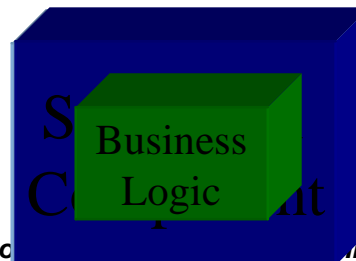
Decoupling SCA logic from business logic

- Provide facilities to connect components
- Supply implementations to work with OE
- Integrate business logic into components



Developer Kits

- BAE SYSTEMS has created Developer Kits based upon many of the patterns and techniques shown in this presentation
- These Developer Kits ease development and further waveform portability
- To a marked degree Waveform and Radio Platform components created using the Developer Kits are shielded from modifications to the SCA itself.
- Business logic can be easily replaced without impact to waveform and radio platform component interfaces (APIs)
- The developer kits creates isolation between the SCA specification and the business logic
 - Allows either to change independently thereby maximizing code re-use and minimizing porting costs



Questions?

Core Framework iPAQ Demo