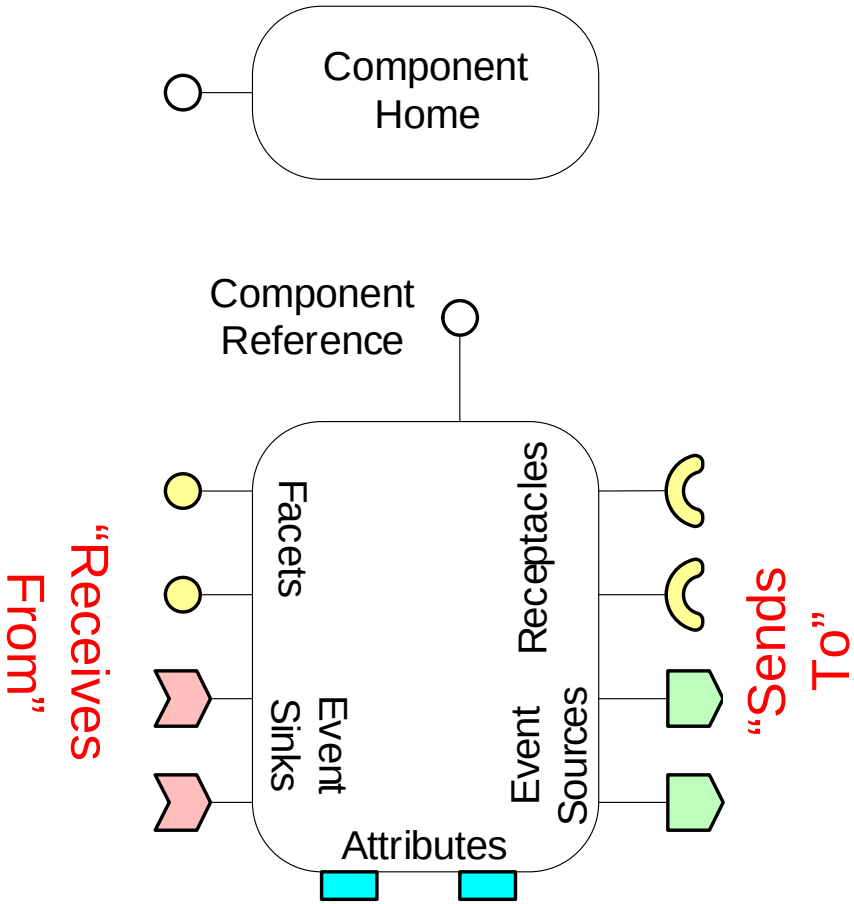
# CORBA Component Ports



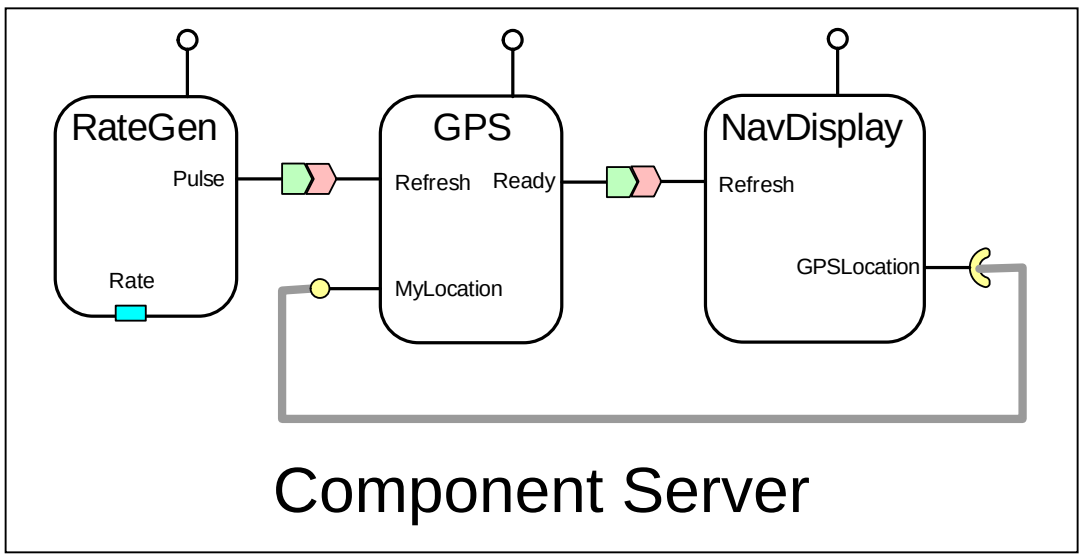

- A CORBA component can contain *ports*:
  - *Facets* (`provides`)
    - Offers operation interfaces
  - *Receptacles* (`uses`)
    - Required operation interfaces
  - *Event sources* (`publishes` & `emits`)
    - Produced events
  - *Event sinks* (`consumes`)
    - Consumed events
  - *Attributes* (`attribute`)
    - configurable properties
- Each component instance is created & managed by a unique component **home**

# Exercise 2: Airplane Application

Rate Generator

Positioning Sensor

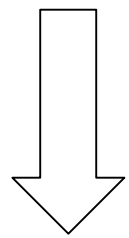
Display Device



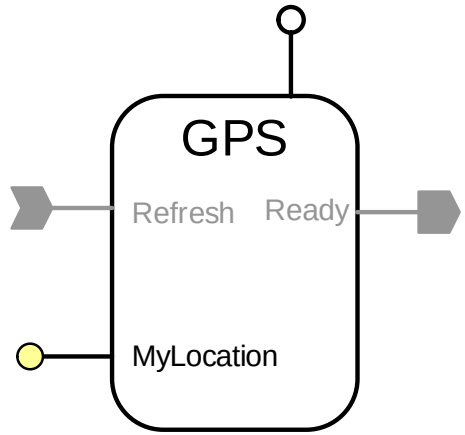

- ***Rate Generator***
  - Sends periodic **`Pulse`** events to consumers
- ***Positioning Sensor***
  - Receives **`Refresh`** events from suppliers
  - Refreshes cached coordinates available thru **`MyLocation`** facet
  - Notifies subscribers via **`Ready`** events
- ***Display Device***
  - Receives **`Refresh`** events from suppliers
  - Reads current coordinates via its **`GPSLocation`** receptacle
  - Updates display

# Component Facets

```
// IDL 3
interface position
{
  long get_pos ();
};
component GPS
{
  provides position MyLocation;
  …
};
```



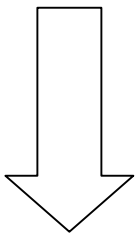
```
// Equivalent IDL 2
interface GPS
  : Components::CCMObject
{
  position
    provide_MyLocation ();
  …
};
```




- Component facets:
  - Define *provided* operation interfaces
  - Specified with **`provides`** keyword
  - Represent the component itself, not a separate thing contained by the component
  - Have independent object references
  - Can be used to implement *Extension Interface* pattern

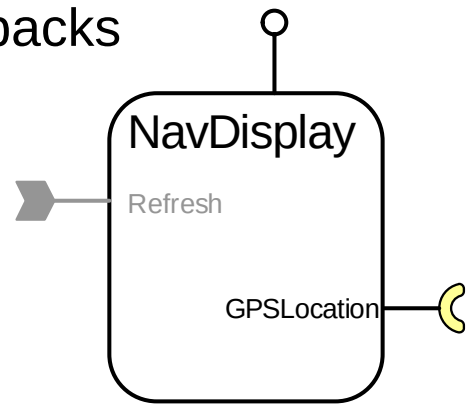
# Component Receptacles

```
// IDL 3
component NavDisplay
{
 …
 uses position GPSLocation;
 …
};
```



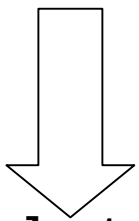
```
// Equivalent IDL 2
interface NavDisplay
  : Components::CCMObject
{
  …
  void connect_GPSLocation
         (in position c);
  position disconnect_GPSLocation();
  position get_connection_GPSLocation ();
  …
};
```

- Component receptacles
  - Specify a way to connect one or more *required* interfaces to this component
  - Specified with **`uses`** keyword
  - Connections are done statically via configuration & deployment tools during initialization stage or assembly stage
  - Dynamically managed at runtime to offer interactions with clients or other components via callbacks

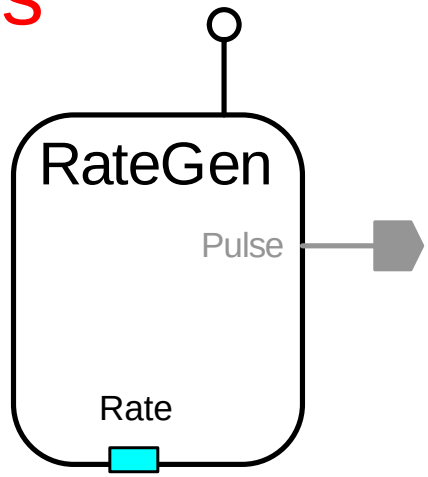


# Component Attributes

```
// IDL 3
typedef unsigned long rateHz;
component RateGen
  supports rate_control
{
  attribute rateHz Rate;
};
```



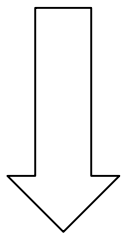
```
// Equivalent IDL 2
interface RateGen :
  Components::CCMObject,
  rate_control
{
  attribute rateHz Rate;
};
```

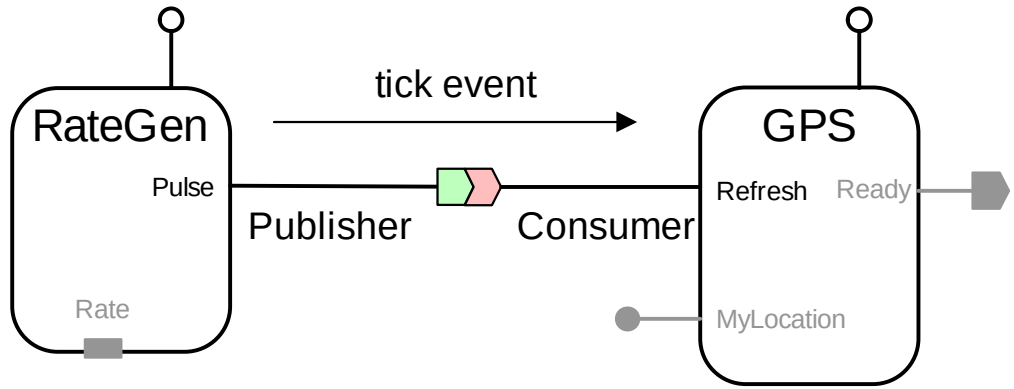



- Named configurable properties
  - Intended for component configuration
    - e.g., optional behaviors, modality, resource hints, etc.
  - Could raise exceptions (new CCM capability)
  - Exposed through accessors & mutators

# Component Events

```
// IDL 3
eventtype tick
{
  public rateHz Rate;
};
```



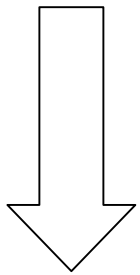



```
// Equivalent IDL 2
valuetype tick : Components::EventBase
{
  public rateHz Rate;
};
interface tickConsumer :
  Components::EventConsumerBase {
  void push_tick
    (in tick the_tick);
};
```

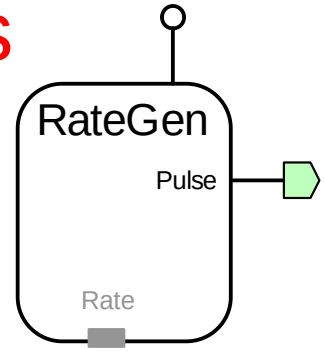
- Events are IDL **`valuetypes`**
- Defined with the new IDL 3 **`eventtype`** keyword

# Component Event Sources

```
// IDL 3
component RateGen
{
  publishes tick Pulse;
  emits tick Trigger;
  …
};
```



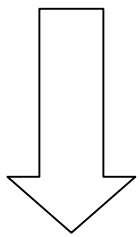
```
// Equivalent IDL 2
interface RateGen :
  Components::CCMObject {
  Components::Cookie
    subscribe_Pulse
    (in tickConsumer c);
  tickConsumer
    unsubscribe_Pulse
    (in Components::Cookie ck);
  …
};
```



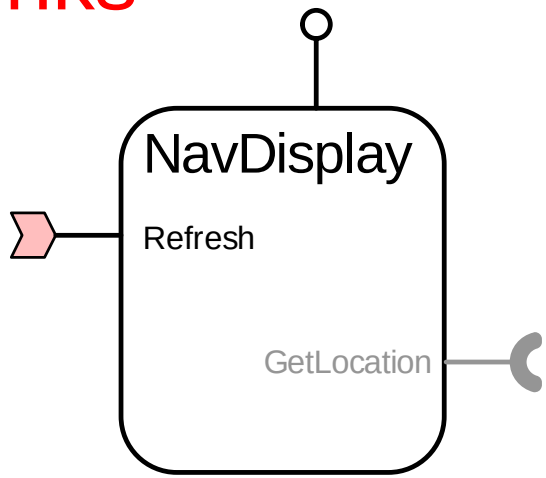

- Event sources
  - Named connection points for event production
  - Two kinds: *publisher & emitter*
    - **`publishes`** = may be multiple consumers
    - **`emits`** = only one consumer
- Event delivery
  - Consumer subscribes/connects directly
  - CCM container mediates access to CosNotification channels or other event delivery mechanism

# Component Event Sinks

```
// IDL 3
component NavDisplay
{
 …
 consumes tick Refresh;
};
```

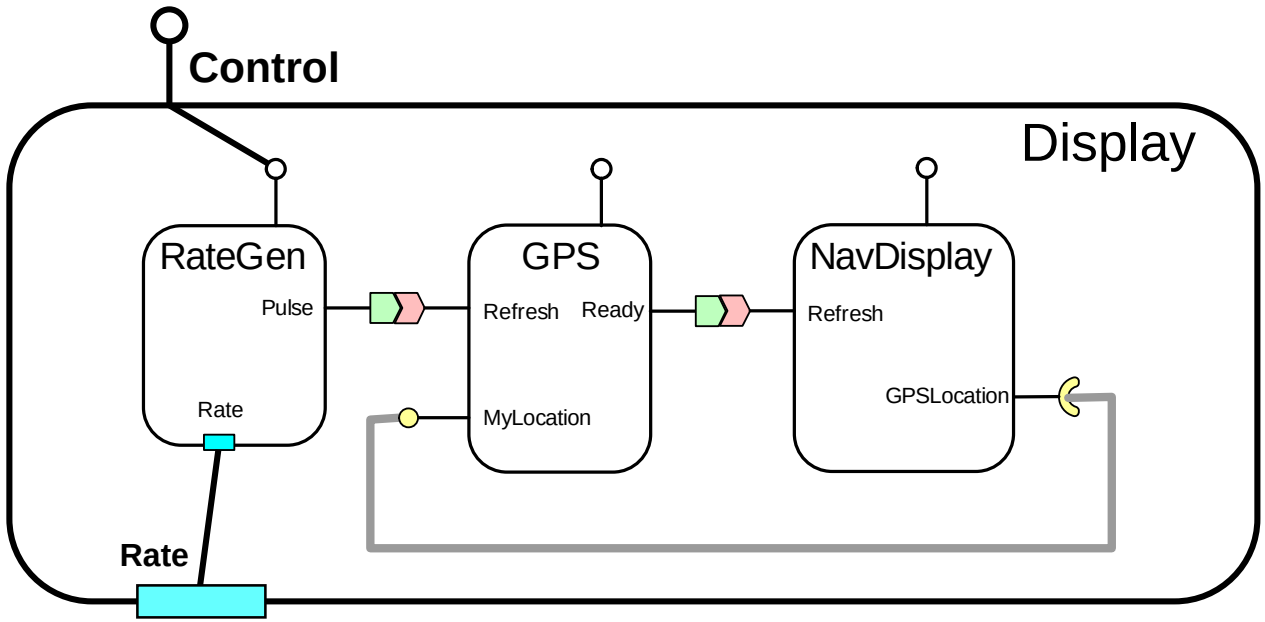


```
// Equivalent IDL 2
interface NavDisplay :
  Components::CCMObject
{
 …
 tickConsumer
   get_consumer_Refresh ();
 …
};
```




- Event sinks
  - Named connection points into which events of a specific type may be pushed
  - Event sink can subscribe to multiple event sources
  - No distinction between emitter & publisher

# Display Component Assembly



```
// IDL 3
eventtype tick{
  public rateHz Rate;
};

interface position{
  long get_pos ();
};

component RateGen{
    attribute rateHz Rate;
    publishes tick Pulse;
};
home RateGenHome manages RateGen {};

component GPS {
        provides position MyLocation;
        consumes tick Refresh;
        publishes tick Ready;
  };
home GPSHome manages GPS {};

component NavDisplay{
        uses position GPSLocation;
        consumes tick Refresh
};
home NavDisplayHome manages NavDisplay {};
```

```
// Equivalent IDL 2
valuetype tick : Components::EventBase{
public rateHz Rate;
};

interface tickConsumer:Components::EventConsumerBase {
void push_tick (in tick the_tick);
};

interface RateGen : Components::CCMObject {

attribute rateHz Rate;

Components::Cookie subscribe_Pulse(in tickConsumer c);
tickConsumer unsubscribe_Pulse(in Components::Cookie ck);
};

interface GPS : Components::CCMObject{

  position provide_MyLocation ();
  tickConsumer get_consumer_Refresh ();
  Components::Cookie subscribe_Ready(in tickConsumer c);
  tickConsumer unsubscribe_Ready(in Components::Cookie ck);

};
```

```
//Equivalent IDL2 (ctd.)

interface NavDisplay : Components::CCMObject{
  void connect_GPSLocation (in position c);
  position disconnect_GPSLocation();
  position get_connection_GPSLocation ();

  tickConsumer get_consumer_Refresh ();
 };
```

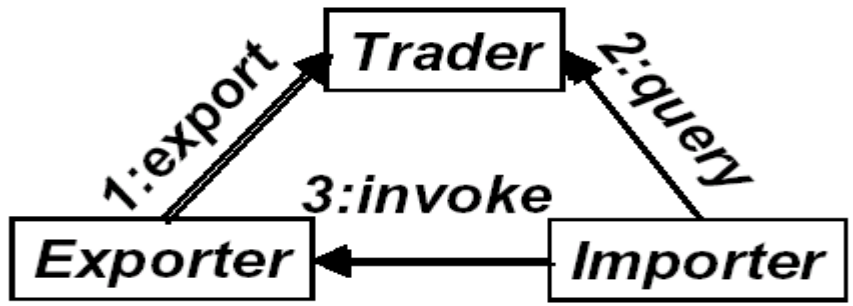
# CORBA Trading Service

## *Rationale*

- Locating objects in location transparent way
- Naming simple but may not be suitable when
  - clients do not know server
  - there are multiple servers to choose from
- Trading supports locating servers based on service functionality and quality
- Naming ⇔ White pages
- Trading ⇔Yellow Pages

# *Trading Operation*

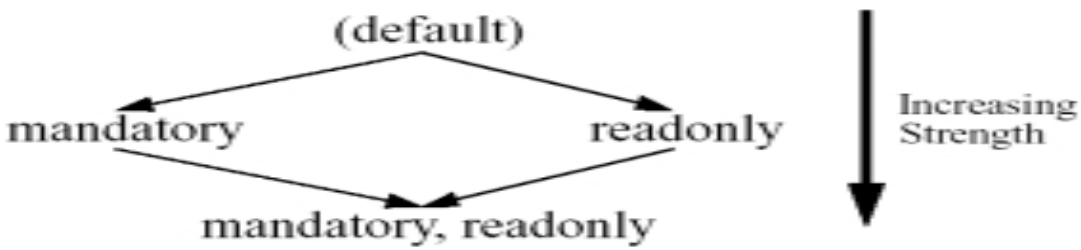
- Trader operates as broker between client and server.
- Enables client to change perspective from ´who?´ to ´what?´
- Similar ideas in:
  - mortgage broker
  - insurance broker




- Clients ask trader for
  - a service of a certain type
  - at a certain level of quality
- Trader supports
  - service matching
  - service shopping
- Server registers service with trader.
- Server defines assured quality of service:
  - Static QoS definition
  - Dynamic QoS definition

# *Service Type Definition*

- Service types define
  - Functionality provided by a service and
  - Qualities of Service (QoS) provision.
- Functionality defined by object type
- QoS defined based on properties, i.e.
  - property name
  - property type
  - property value
  - property mode
    - mandatory/optional
    - readonly/modifiable



# Exercise 3: Telephone Service

**Factors affecting the Load balancing**

-Phone servers register with Trader by specifying their service type
-Client query the Trader by specifying their needs
-Trader matches clients needs with available services

Phone service: location, VoIP protocol,Price, Link quality

```
typedef enum {MGCP, H323, SIP} Protocol;
service internet_phone {
        interface PhoneServer;
        readonly mandatory property long location;
        readonly mandatory property Protocol voip_prot;
         readonly mandatory property double price;
        readonly mandatory property LinkQuality quality;
}
```

Client: area code, destination number, VoIP protocol, service plan

# Call Scenario