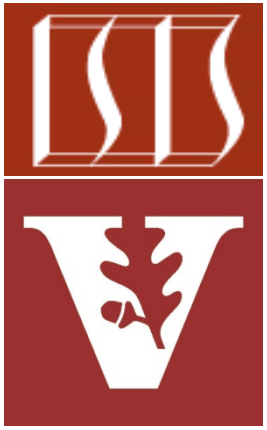
Patterns & Frameworks for Synchronous Event Handling, Connections, & Service Initialization: Part 1

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt



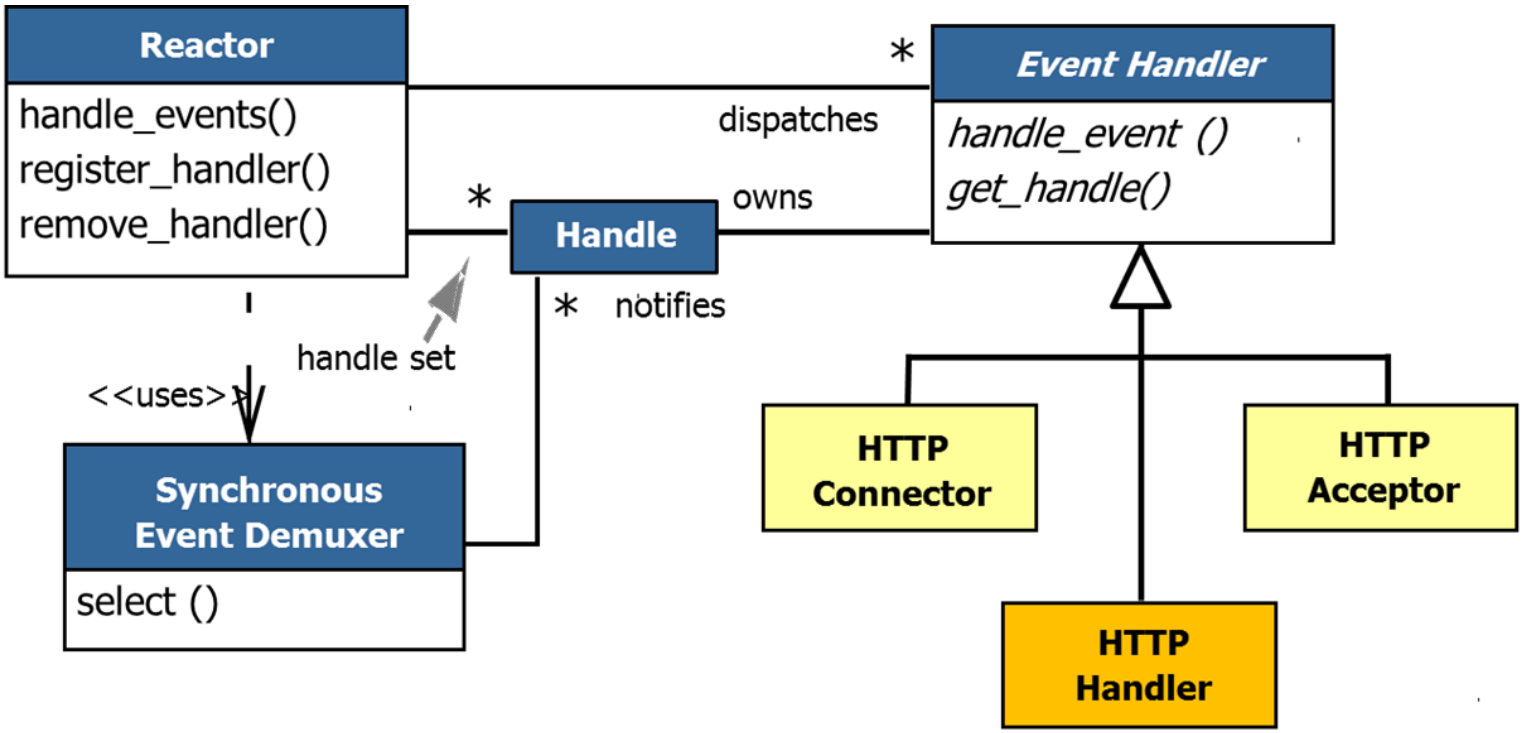
Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Topics Covered in this Part of the Module

- Describe the *Reactor* & *Acceptor-Connector* patterns



Simplifying Event Handling Concerns

Context	Problem
• Web servers must demux & process multiple types of indication events arriving from clients simultaneously	• Developers often tightly couple event demuxing & connection code with app protocol processing code, which impedes understanding, reuse, & optimization

Use of non-portable handle type

Hard-coded to a particular system call

Inefficient O(n) sequential search

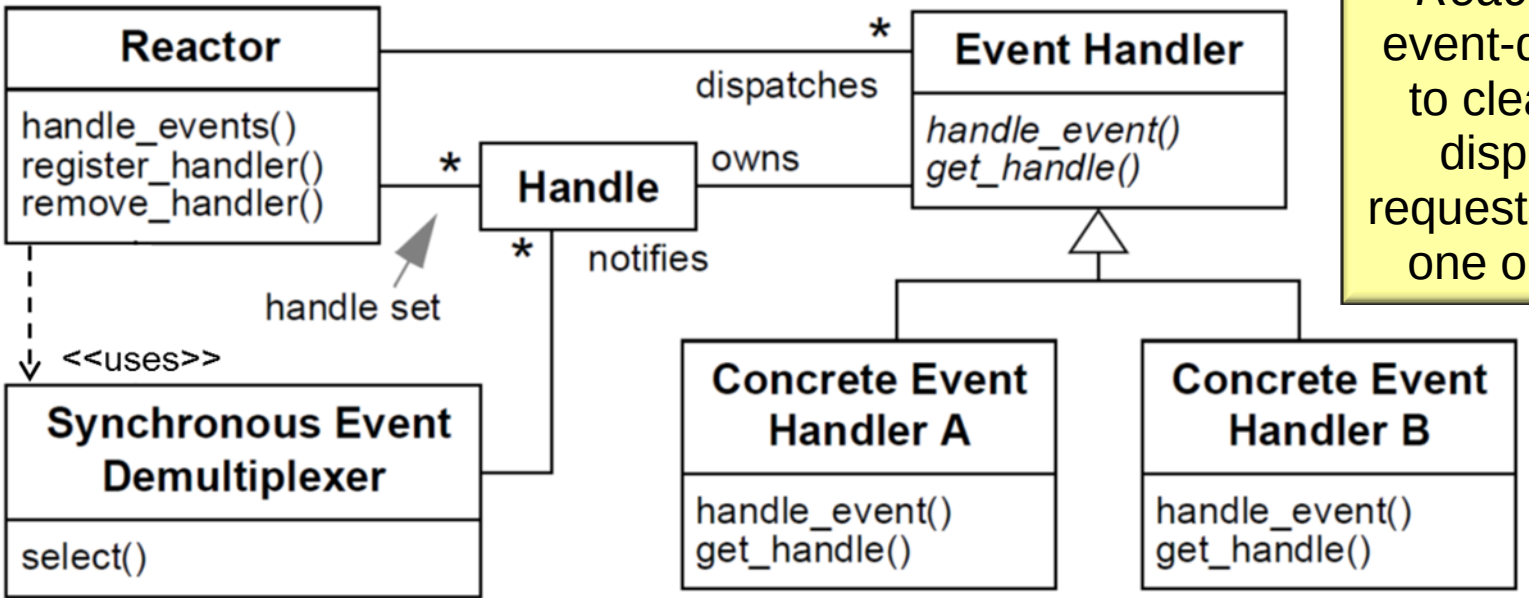
Tightly-coupled solution

```
select (width, &ready_handles, 0, 0, 0);
if (FD_ISSET (acceptor,  &ready_handles)) {
   int h = accept (acceptor, 0, 0);
   FD_SET (handle, &ready_handles);
   // ... Handle connection acceptance
} else
   for (int handle = s_handle + 1;
        handle < maxhandlep1; ++handle)
     if (FD_ISSET (handle, &ready_handles)) {
       // ... Perform HTTP processing
     }
```

Simplifying Event Handling Concerns

Context	Problem	Solution
• Web servers must demux & process multiple types of indication events arriving from clients simultaneously	• Developers often tightly couple event demuxing & connection code with app protocol processing code, which impedes understanding, reuse, & optimization	• Apply ***Reactor*** pattern & ***Acceptor-Connector*** pattern to separate the generic event demuxing & connection code from the JAWS protocol code

Structure

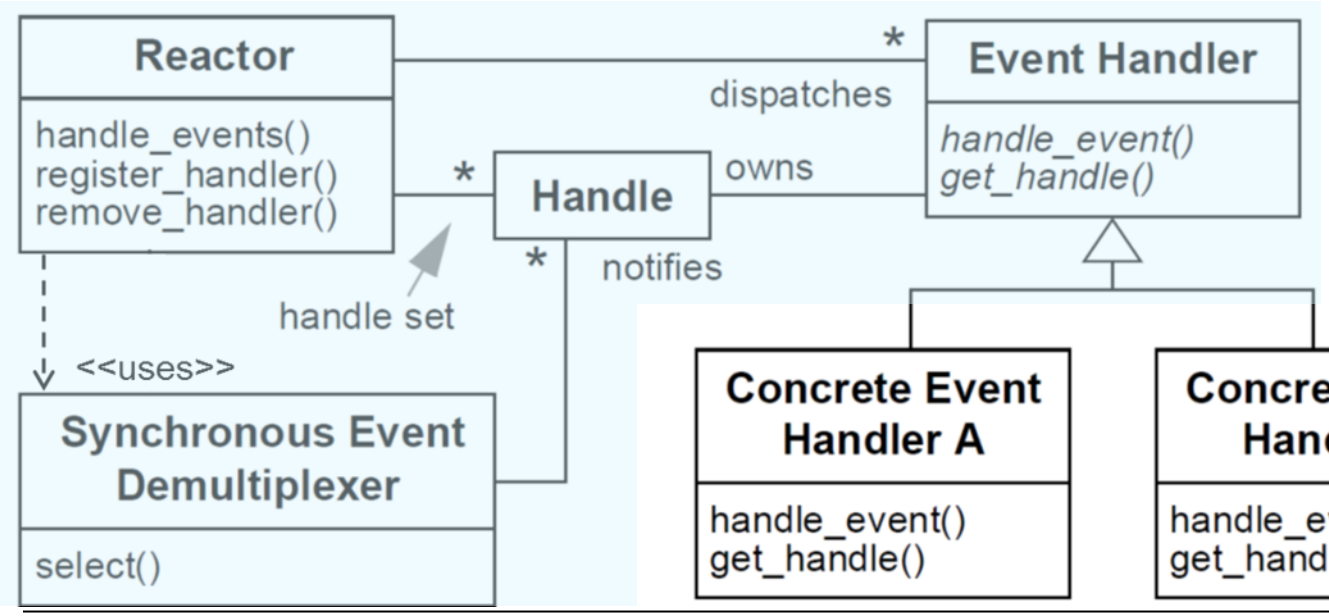


Reactor structures event-driven software to cleanly demux & dispatch service requests received from one or more clients

Simplifying Event Handling Concerns

Context	Problem	Solution
• Web servers must demux & process multiple types of indication events arriving from clients simultaneously	• Developers often tightly couple event demuxing & connection code with app protocol processing code, which impedes understanding, reuse, & optimization	• Apply ***Reactor*** pattern & ***Acceptor-Connector*** pattern to separate the generic event demuxing & connection code from the JAWS protocol code

Structure

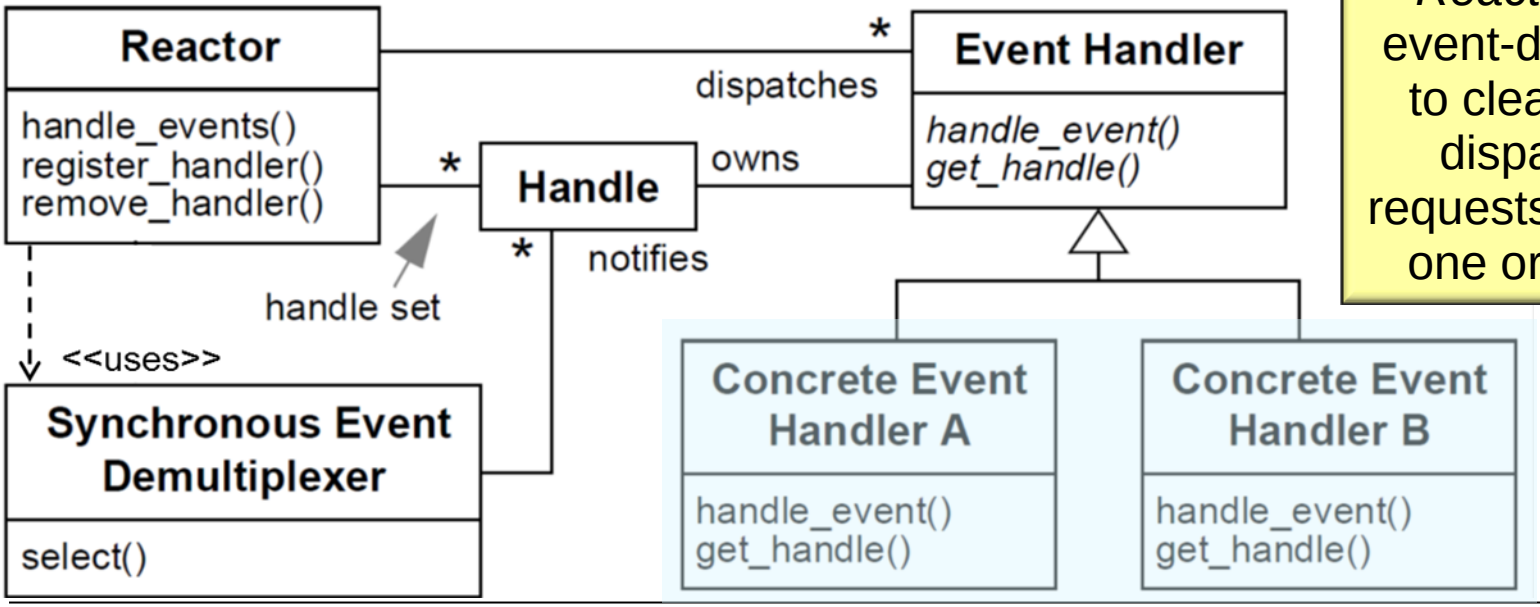


Reactor structures event-driven software to cleanly demux & dispatch service requests received from one or more clients

Simplifying Event Handling Concerns

Context	Problem	Solution
• Web servers must demux & process multiple types of indication events arriving from clients simultaneously	• Developers often tightly couple event demuxing & connection code with app protocol processing code, which impedes understanding, reuse, & optimization	• Apply ***Reactor*** pattern & ***Acceptor-Connector*** pattern to separate the generic event demuxing & connection code from the JAWS protocol code

Structure



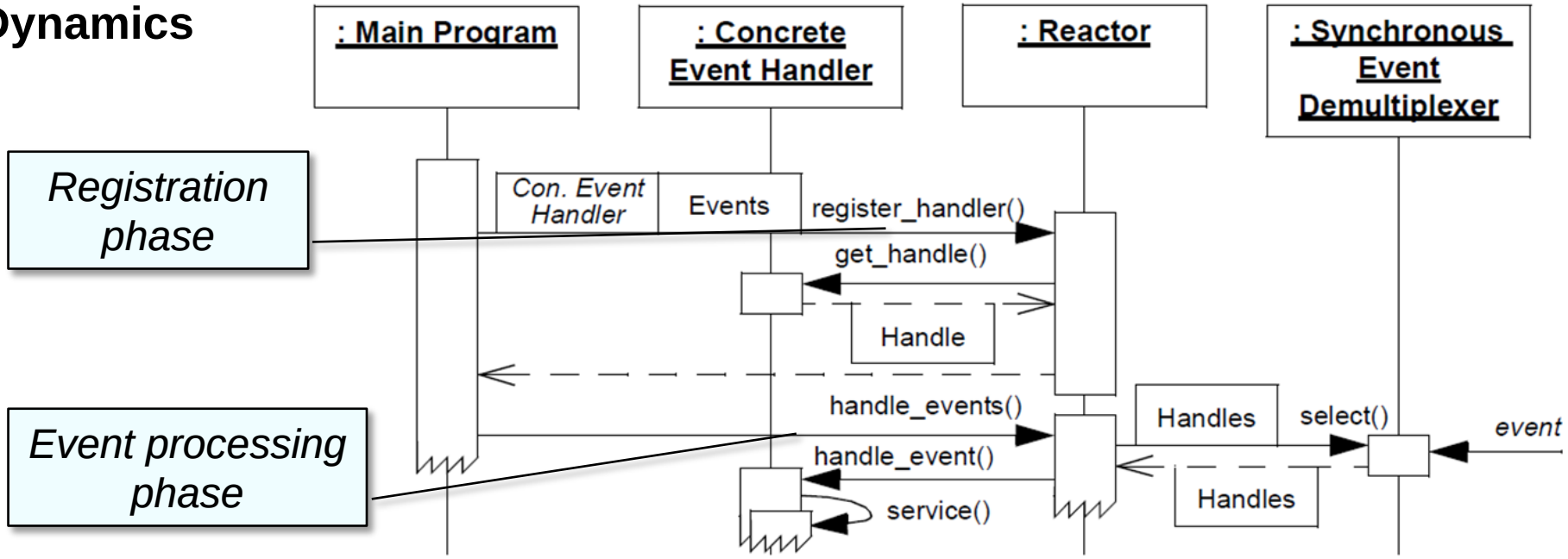
Reactor structures event-driven software to cleanly demux & dispatch service requests received from one or more clients

See www.dre.vanderbilt.edu/~schmidt/PDF/Reactor.pdf for more info

Simplifying Event Handling Concerns

Context	Problem	Solution
• Web servers must demux & process multiple types of indication events arriving from clients simultaneously	• Developers often tightly couple event demuxing & connection code with app protocol processing code, which impedes understanding, reuse, & optimization	• Apply ***Reactor*** pattern & ***Acceptor-Connector*** pattern to separate the generic event demuxing & connection code from the JAWS protocol code

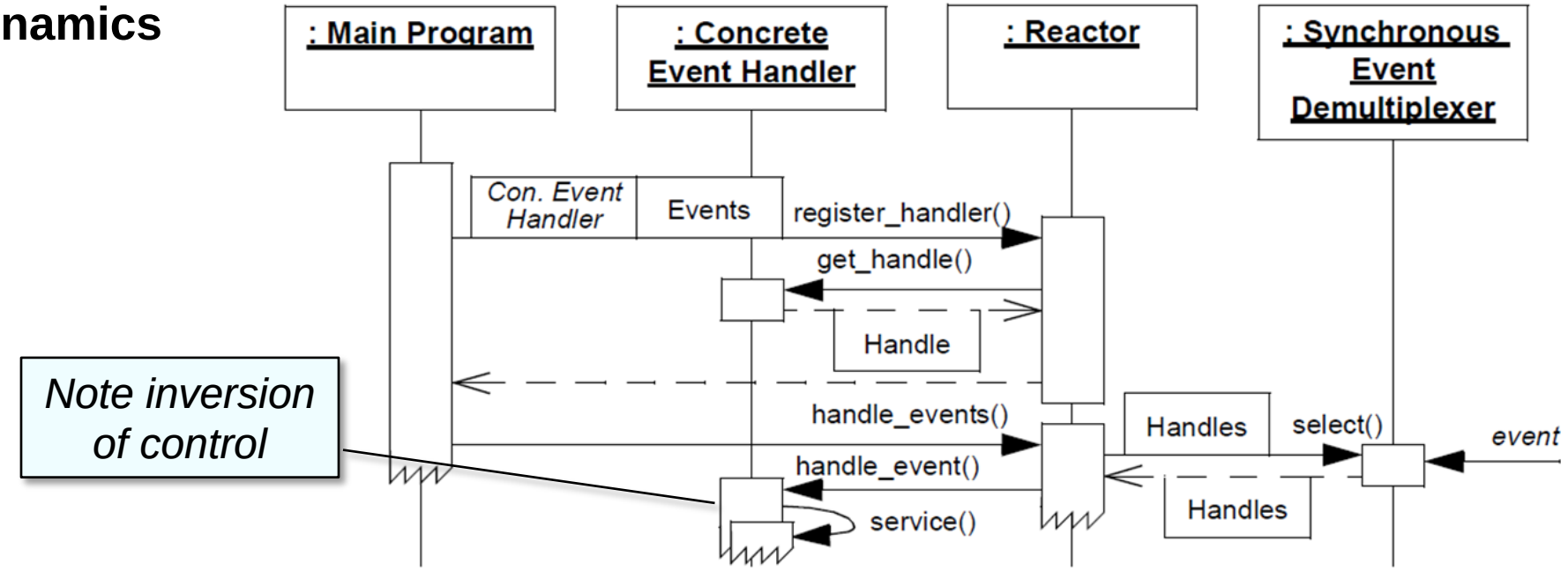
Dynamics



Simplifying Event Handling Concerns

Context	Problem	Solution
• Web servers must demux & process multiple types of indication events arriving from clients simultaneously	• Developers often tightly couple event demuxing & connection code with app protocol processing code, which impedes understanding, reuse, & optimization	• Apply ***Reactor*** pattern & ***Acceptor-Connector*** pattern to separate the generic event demuxing & connection code from the JAWS protocol code

Dynamics

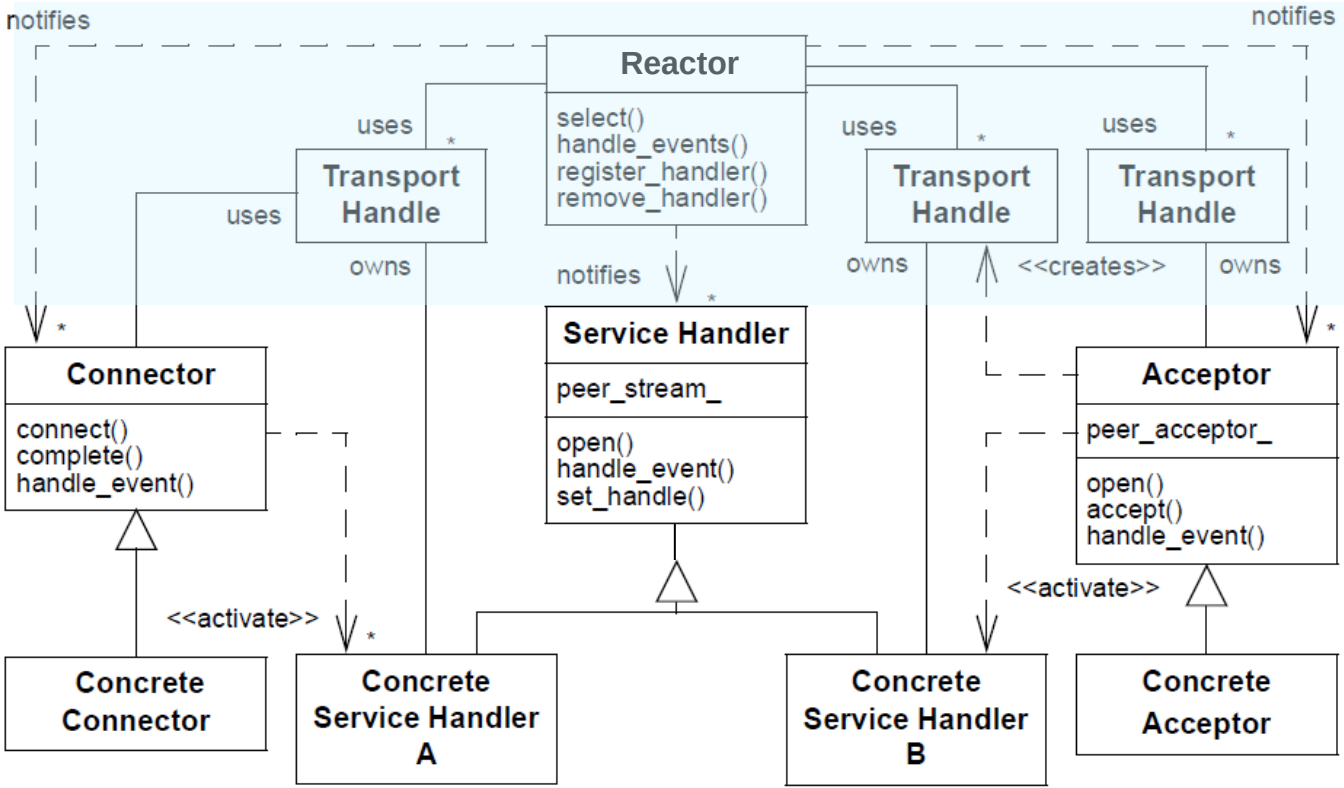


Simplifying Event Handling Concerns

Context	Problem	Solution
• Web servers must demux & process multiple types of indication events arriving from clients simultaneously	• Developers often tightly couple event demuxing & connection code with app protocol processing code, which impedes understanding, reuse, & optimization	• Apply ***Reactor*** pattern & ***Acceptor-Connector*** pattern to separate the generic event demuxing & connection code from the JAWS protocol code

Structure

Acceptor-Connector decouples the connection & initialization of peer services from the processing performed by the peer services after being connected & initialized

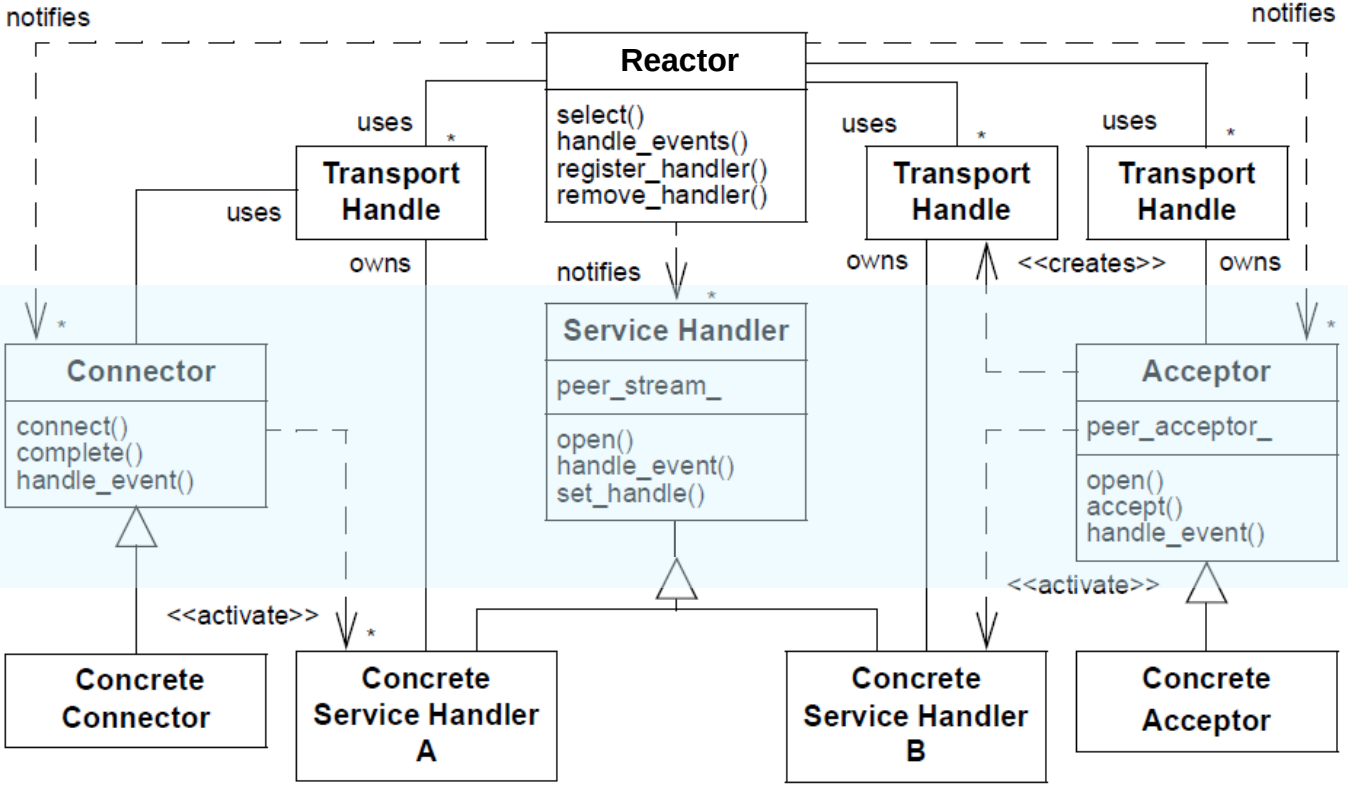


Simplifying Event Handling Concerns

Context	Problem	Solution
• Web servers must demux & process multiple types of indication events arriving from clients simultaneously	• Developers often tightly couple event demuxing & connection code with app protocol processing code, which impedes understanding, reuse, & optimization	• Apply ***Reactor*** pattern & ***Acceptor-Connector*** pattern to separate the generic event demuxing & connection code from the JAWS protocol code

Structure

Acceptor-Connector decouples the connection & initialization of peer services from the processing performed by the peer services after being connected & initialized

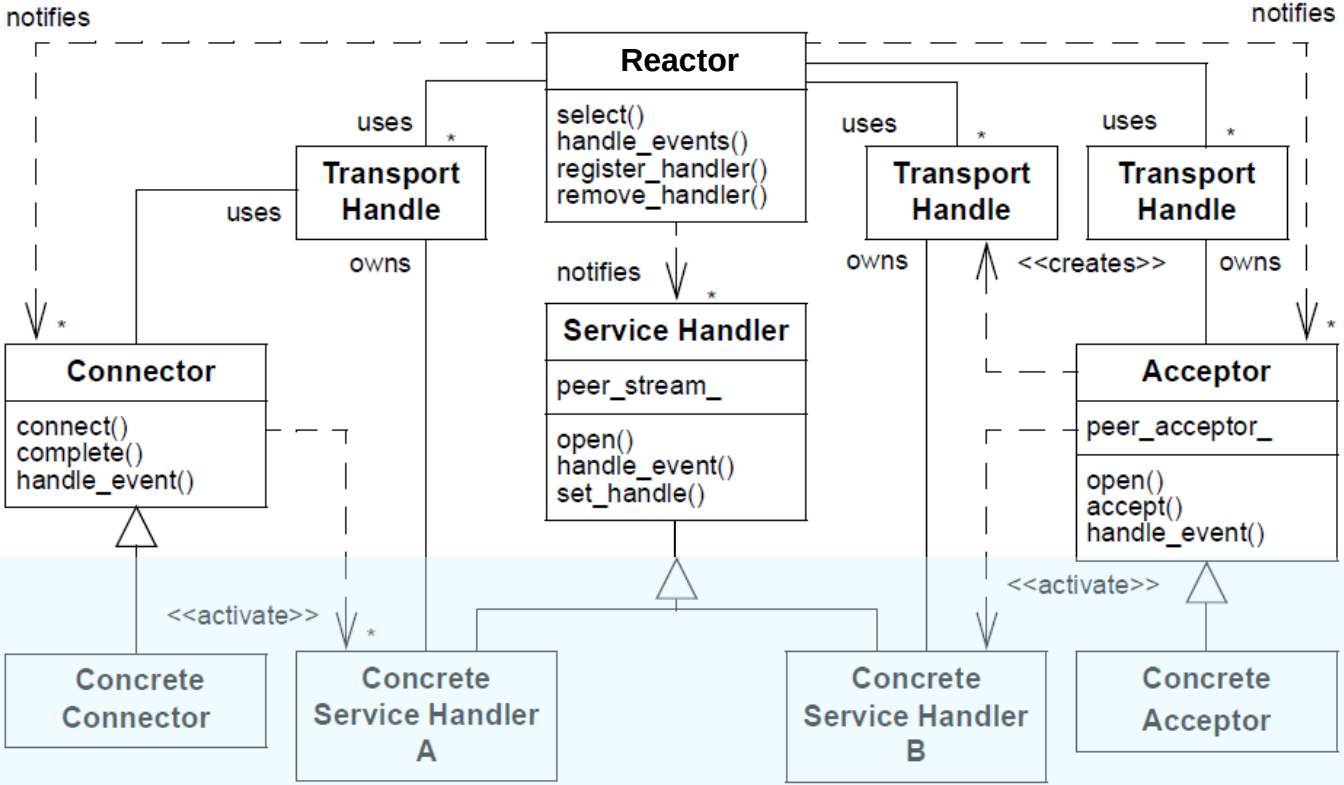


Simplifying Event Handling Concerns

Context	Problem	Solution
• Web servers must demux & process multiple types of indication events arriving from clients simultaneously	• Developers often tightly couple event demuxing & connection code with app protocol processing code, which impedes understanding, reuse, & optimization	• Apply ***Reactor*** pattern & ***Acceptor-Connector*** pattern to separate the generic event demuxing & connection code from the JAWS protocol code

Structure

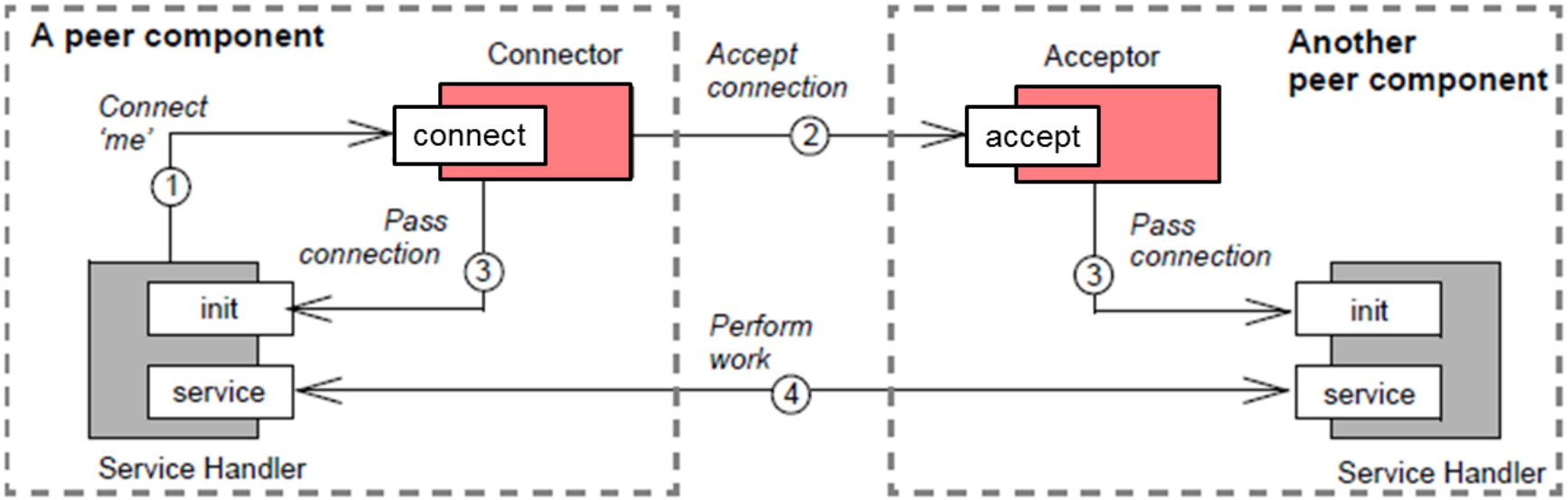
Acceptor-Connector decouples the connection & initialization of peer services from the processing performed by the peer services after being connected & initialized



Simplifying Event Handling Concerns

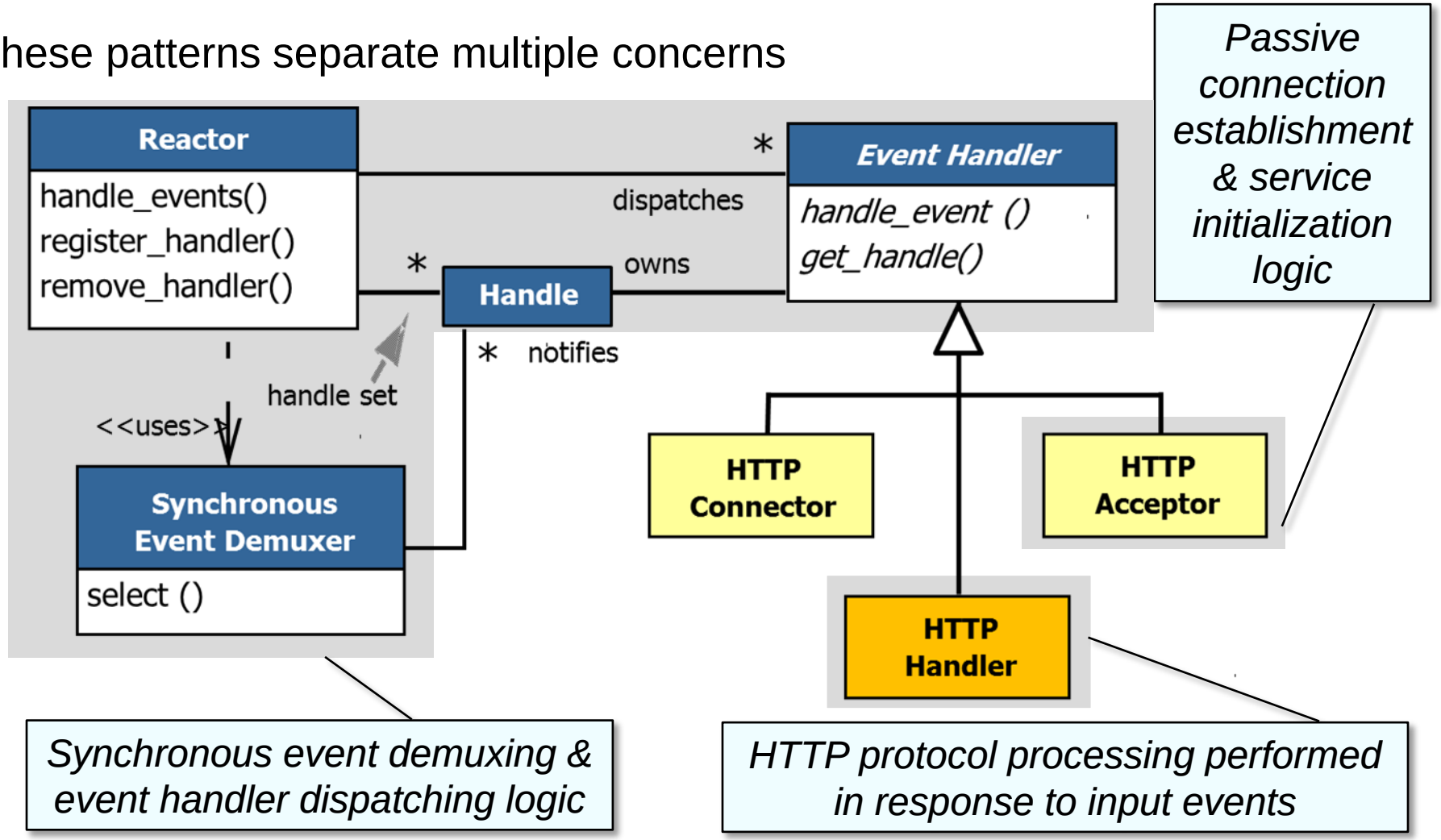
Context	Problem	Solution
• Web servers must demux & process multiple types of indication events arriving from clients simultaneously	• Developers often tightly couple event demuxing & connection code with app protocol processing code, which impedes understanding, reuse, & optimization	• Apply ***Reactor*** pattern & ***Acceptor-Connector*** pattern to separate the generic event demuxing & connection code from the JAWS protocol code

Dynamics



Applying the Reactor & Acceptor-Connector Patterns to JAWS

These patterns separate multiple concerns

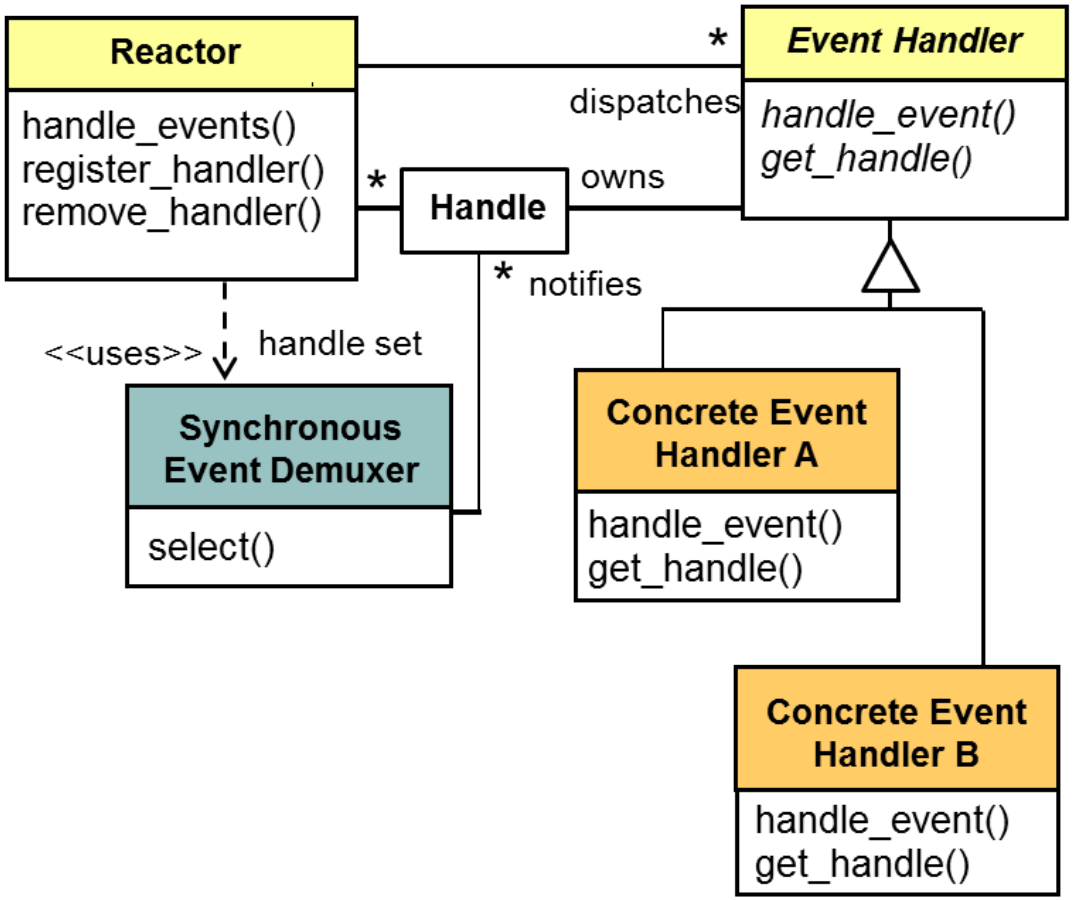


JAWS doesn't use *Connector* portion of the *Acceptor-Connector* pattern

Benefits of the Reactor Pattern

Separation of concerns

- Decouples generic demuxing & dispatching mechanisms from app-specific functionality



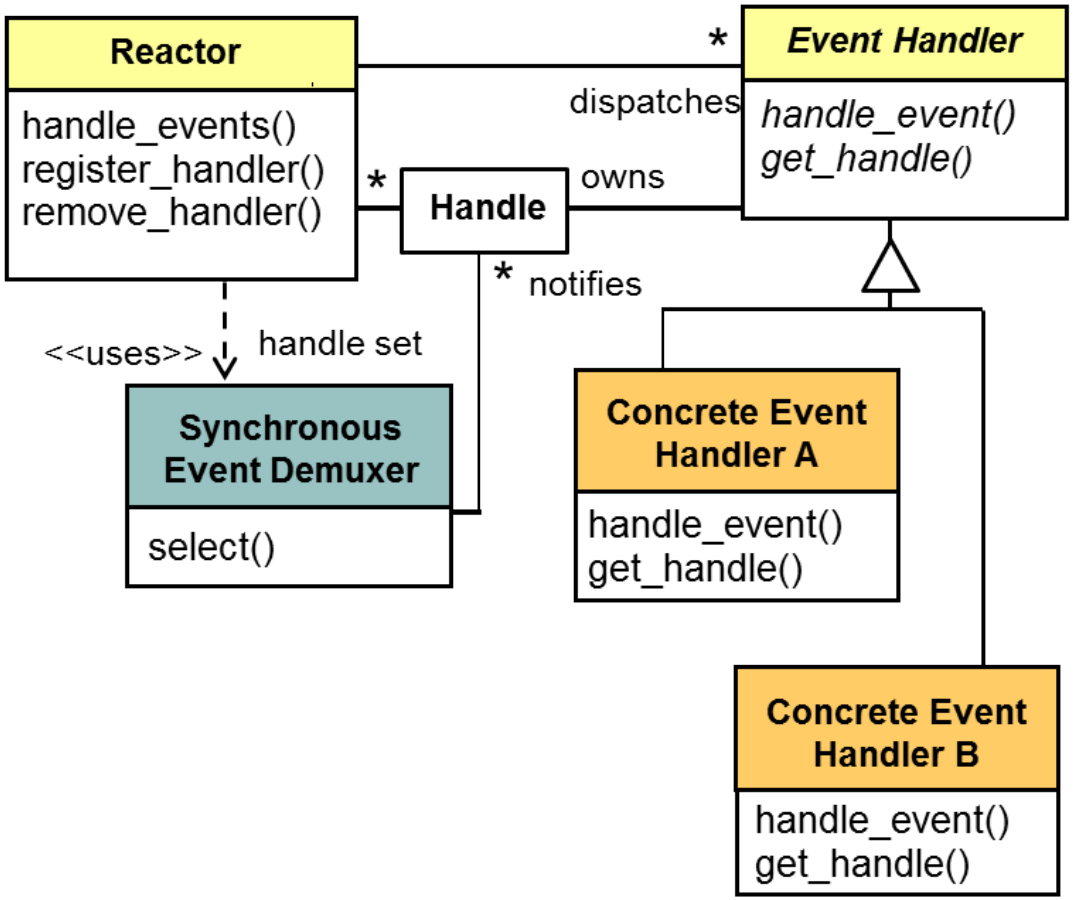
Benefits of the Reactor Pattern

Separation of concerns

- Decouples generic demuxing & dispatching mechanisms from app-specific functionality

Modularity, reusability, & configurability

- Separates event-driven app functionality into several loosely coupled components



Benefits of the Reactor Pattern

Separation of concerns

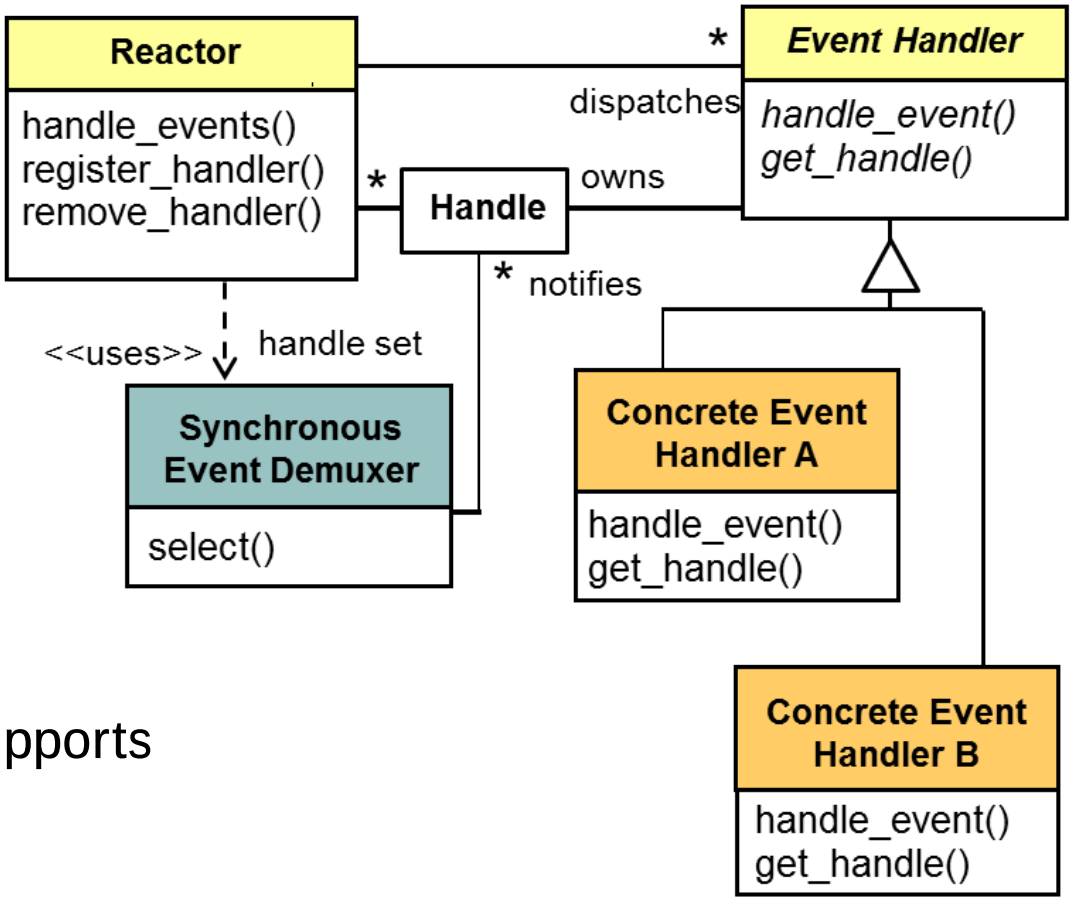
- Decouples generic demuxing & dispatching mechanisms from app-specific functionality

Modularity, reusability, & configurability

- Separates event-driven app functionality into several loosely coupled components

Portability

- Can run on any platform that supports synchronous muxers



Benefits of the Reactor Pattern

Separation of concerns

- Decouples generic demuxing & dispatching mechanisms from app-specific functionality

Modularity, reusability, & configurability

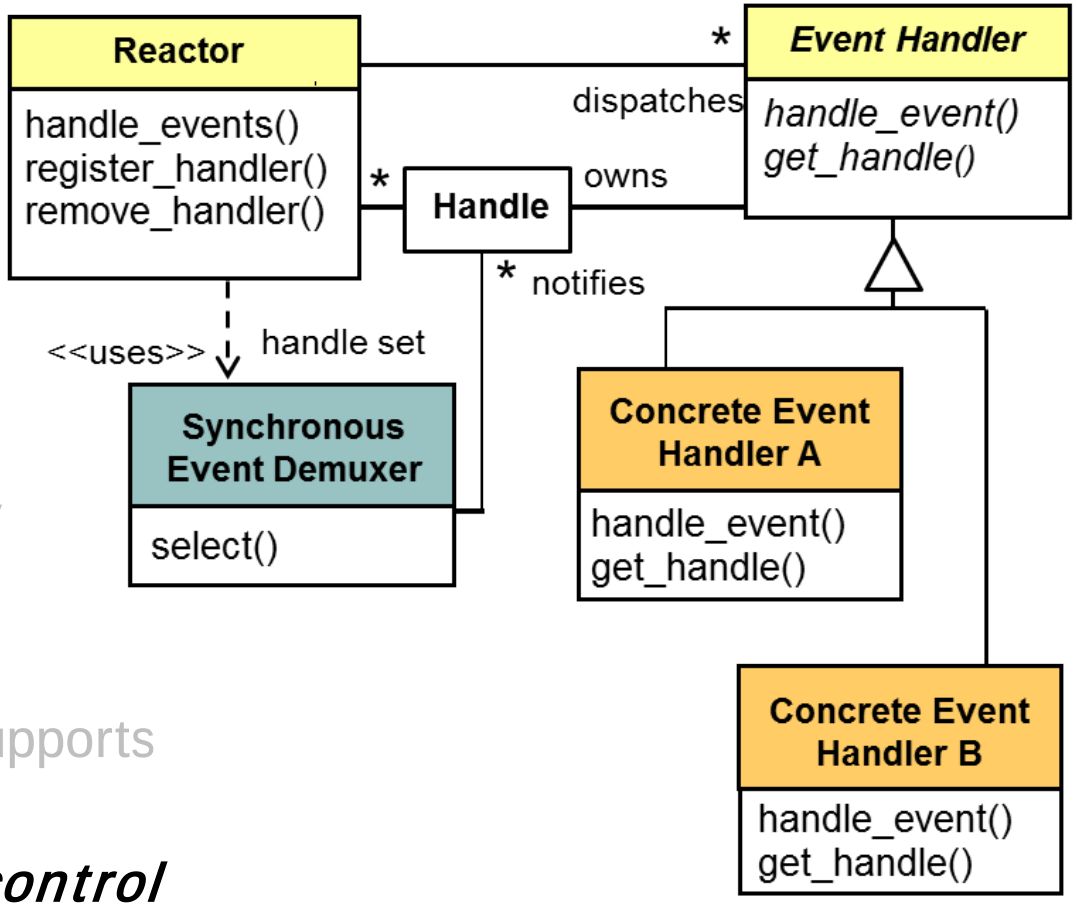
- Separates event-driven app functionality into several loosely coupled components

Portability

- Can run on any platform that supports synchronous muxers

Coarse-grained concurrency control

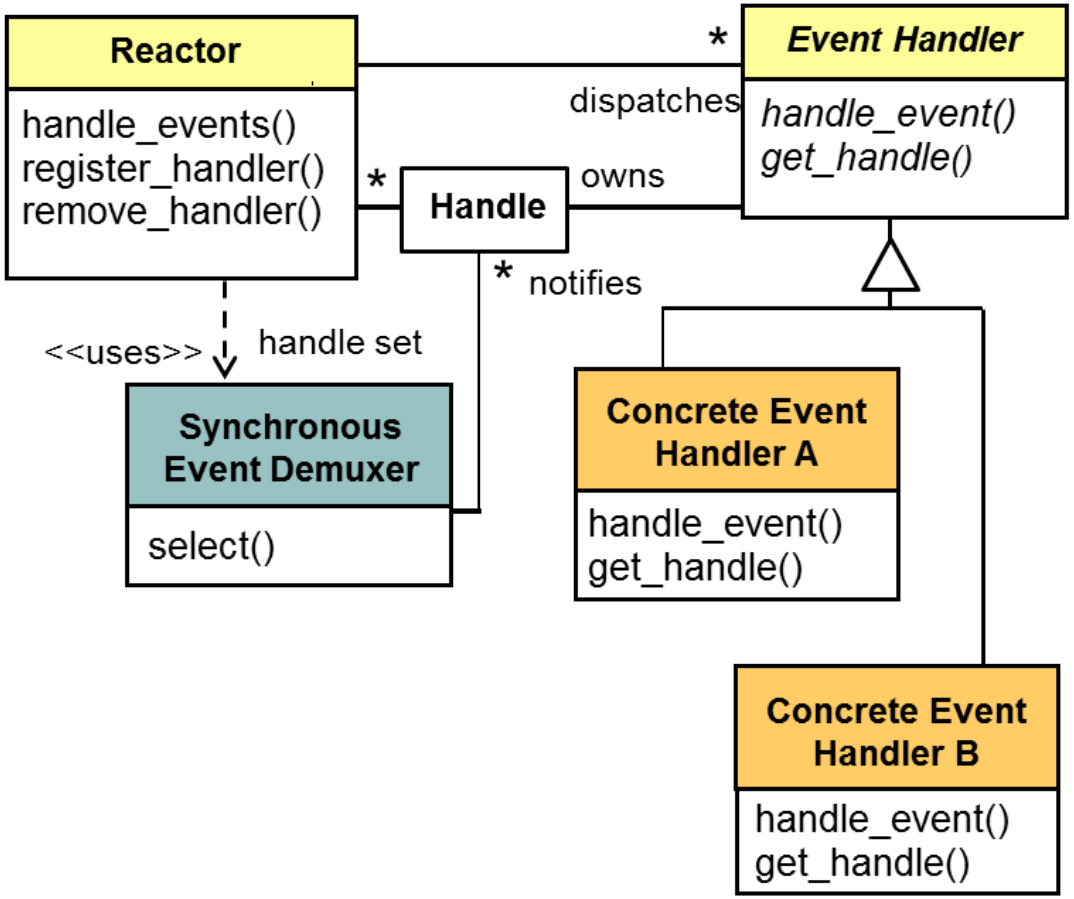
- Serializes the invocation of event handlers at the level of event demuxing & dispatching within an application process



Limitations of the Reactor Pattern

Restricted applicability

- Can be applied efficiently only if OS supports synchronous event demuxing



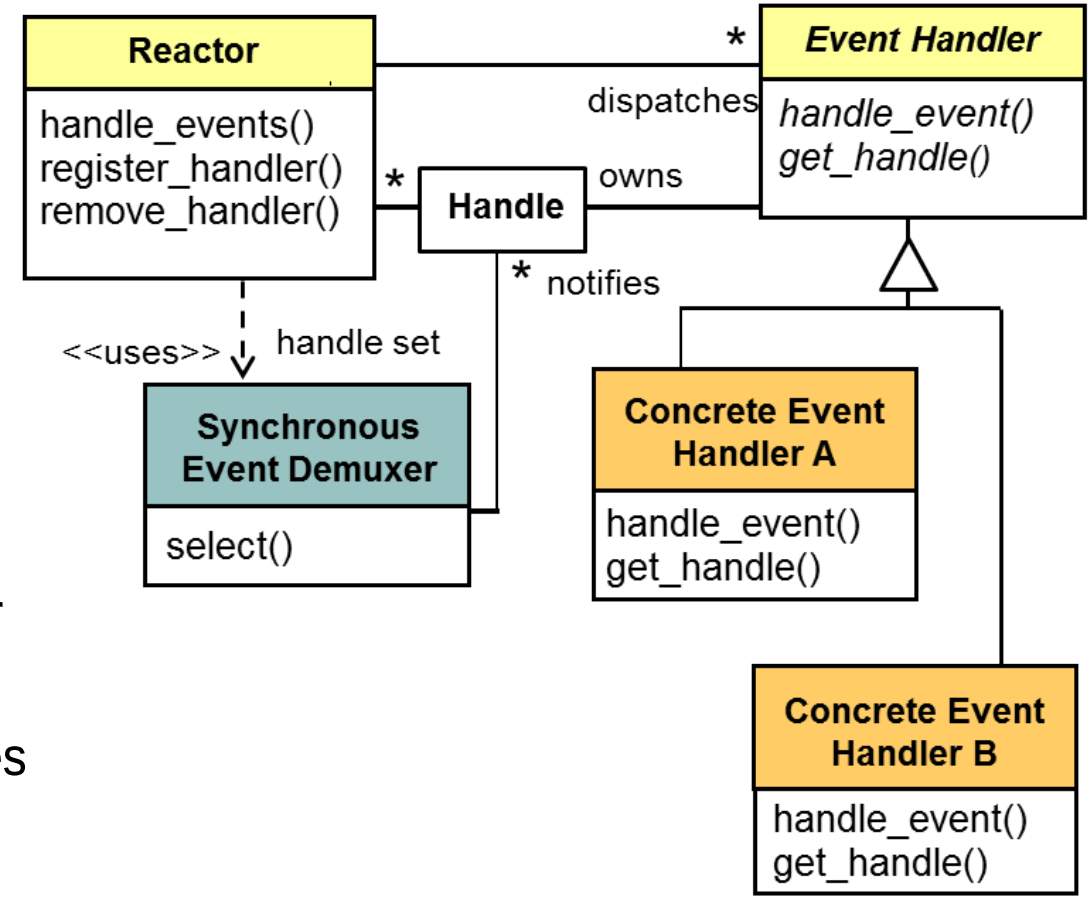
Limitations of the Reactor Pattern

Restricted applicability

- Can be applied efficiently only if OS supports synchronous event demuxing

Non-pre-emptive & non-scalable

- Event handlers borrow reactor thread & run to completion, preventing other event handler dispatching
- Can't leverage multi-CPUs/cores



Limitations of the Reactor Pattern

Restricted applicability

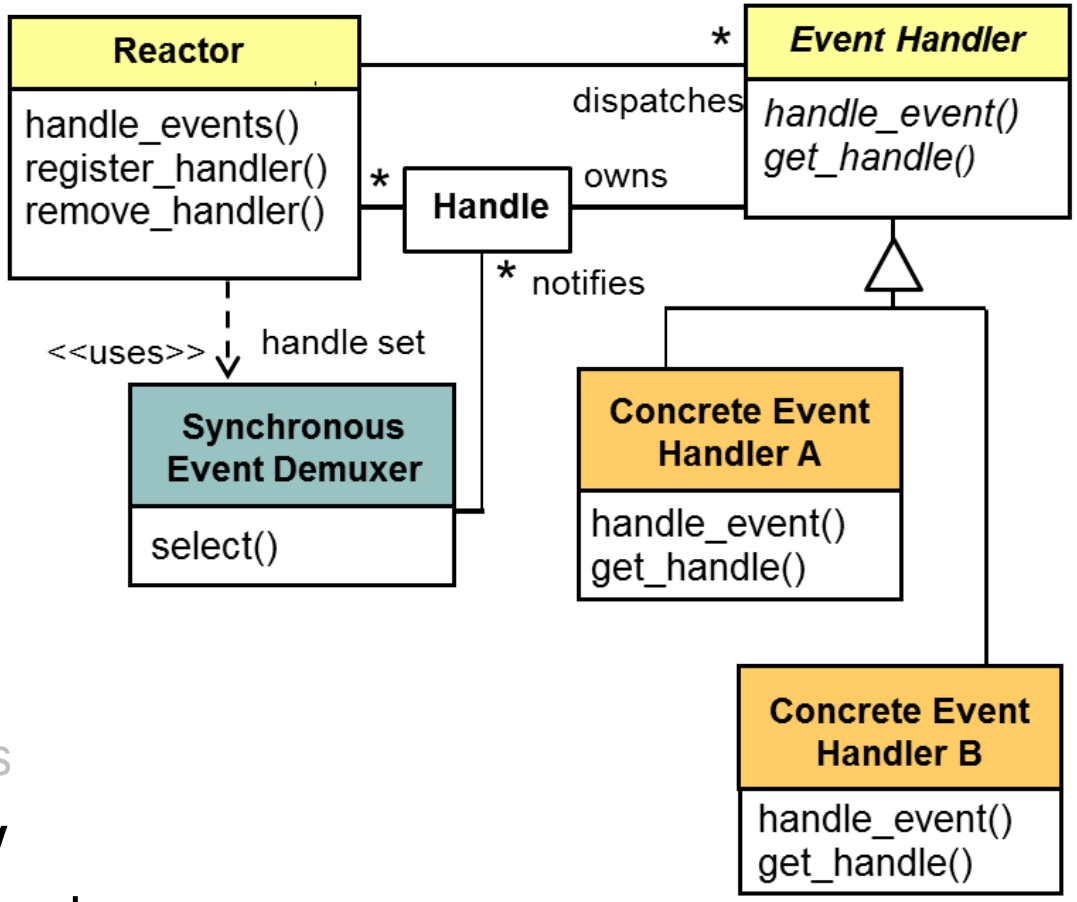
- Can be applied efficiently only if OS supports synchronous event demuxing

Non-pre-emptive & non-scalable

- Event handlers borrow reactor thread & run to completion, preventing other event handler dispatching
- Can't leverage multi-CPUs/cores

Complex debugging & testing

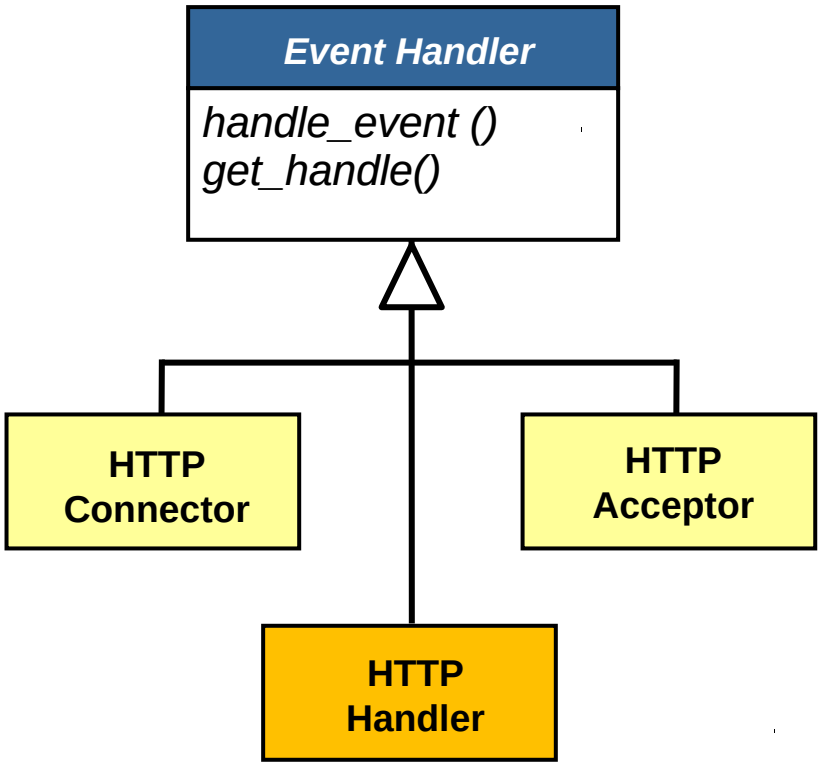
- It is hard to debug apps structured using inversion of control, which oscillates between framework infrastructure & method call-backs on app-specific event handlers



Benefits of the Acceptor-Connector Pattern

Reusability, portability, & extensibility

- Decouples mechanisms for connecting & initializing service handlers from the service processing performed after service handlers are connected & initialized



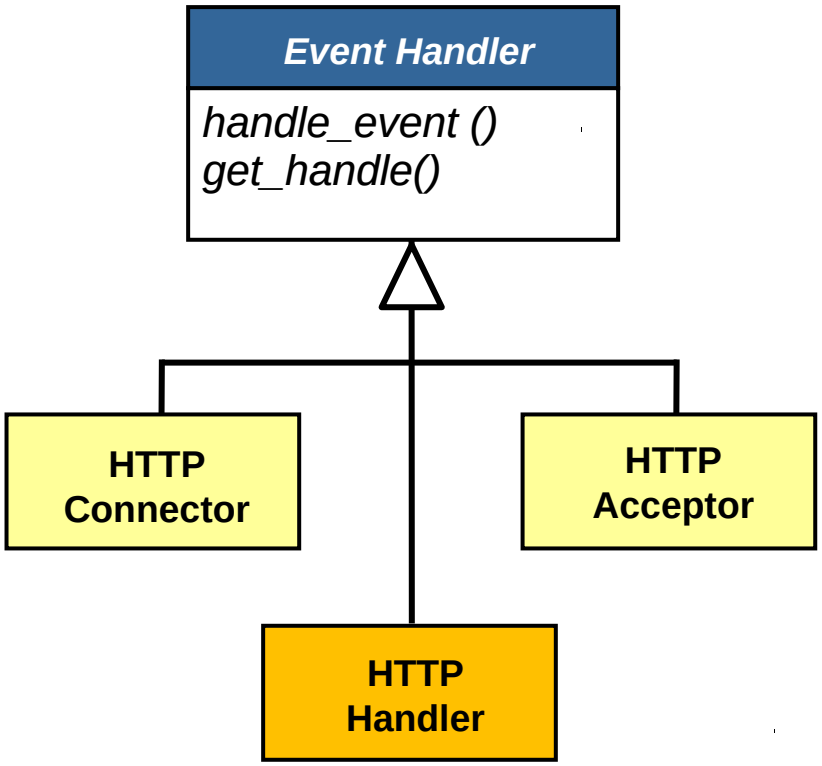
Benefits of the Acceptor-Connector Pattern

Reusability, portability, & extensibility

- Decouples mechanisms for connecting & initializing service handlers from the service processing performed after service handlers are connected & initialized

Robustness

- Strongly decouples the service handler from acceptor & connector, which ensures that a data-mode transport endpoint can't accidentally be used to accept or connect



Benefits of the Acceptor-Connector Pattern

Reusability, portability, & extensibility

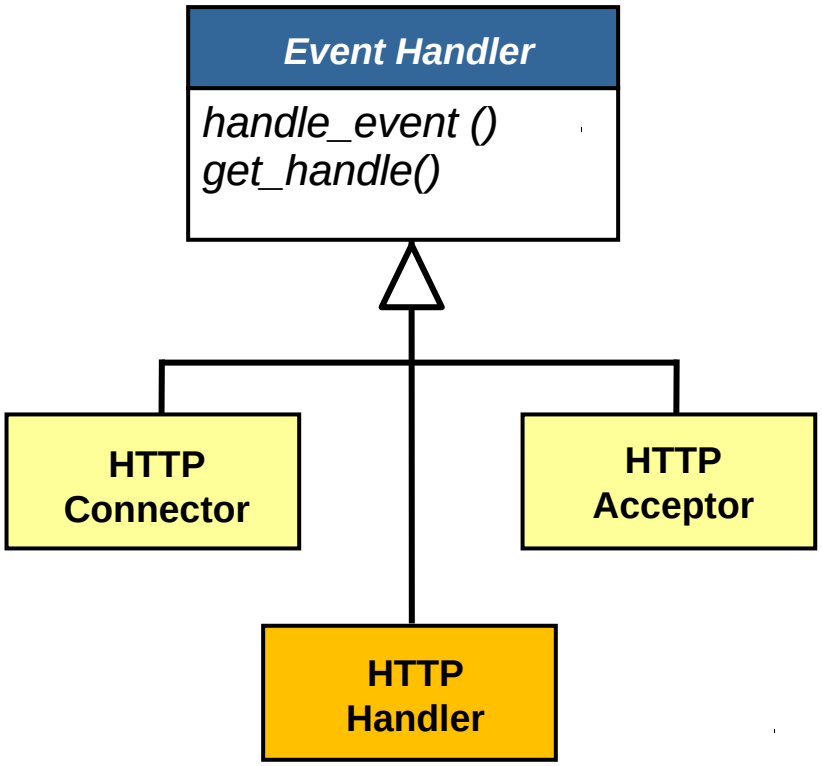
- Decouples mechanisms for connecting & initializing service handlers from the service processing performed after service handlers are connected & initialized

Robustness

- Strongly decouples the service handler from acceptor & connector, which ensures that a data-mode transport endpoint can't accidentally be used to accept or connect

Efficiency

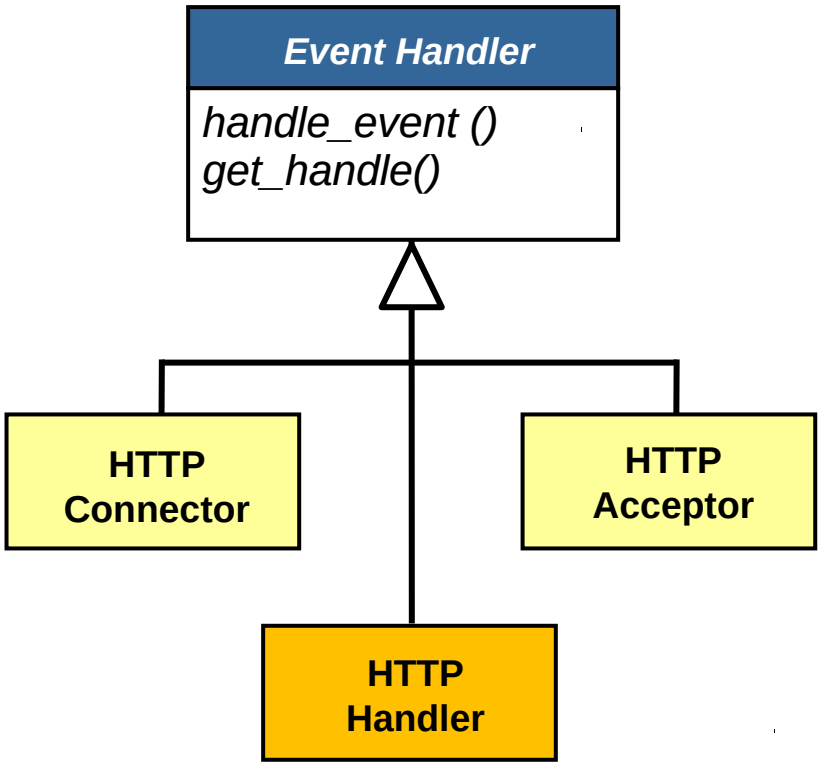
- Can establish connections actively with many hosts asynchronously & efficiently over long-latency wide area networks



Limitations of the Acceptor-Connector Pattern

Additional indirection

- Can incur additional indirection compared to using the underlying network programming interfaces directly



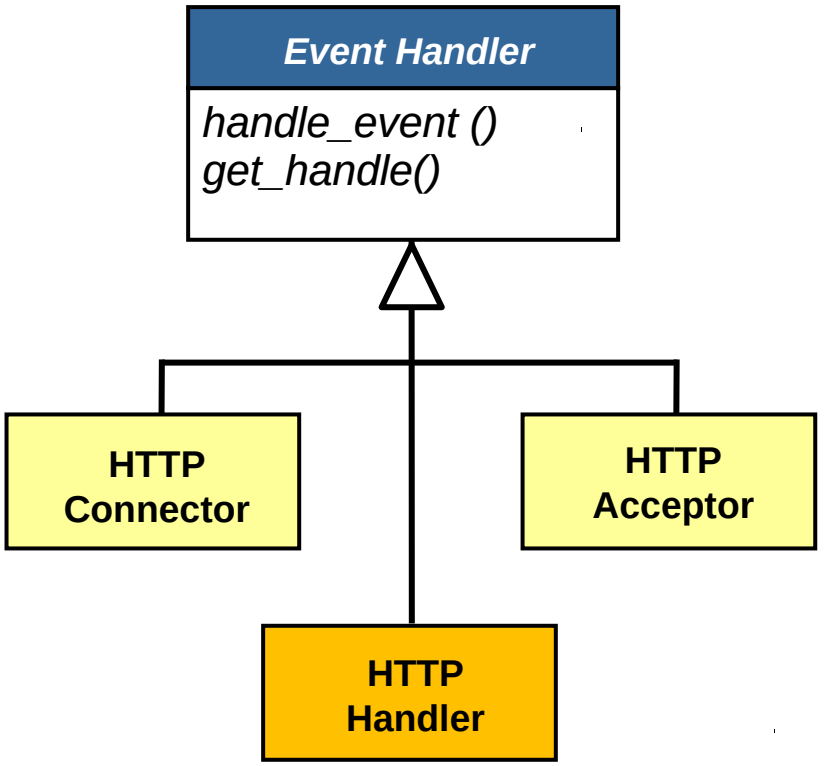
Limitations of the Acceptor-Connector Pattern

Additional indirection

- Can incur additional indirection compared to using the underlying network programming interfaces directly

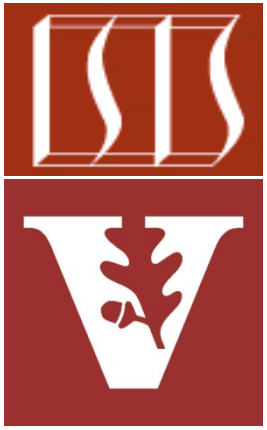
Additional complexity

- May add unnecessary complexity for simple client applications that connect with only one server & perform one service using a single network programming interface



Patterns & Frameworks for Synchronous Event Handling, Connections, & Service Initialization: Part 2

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt



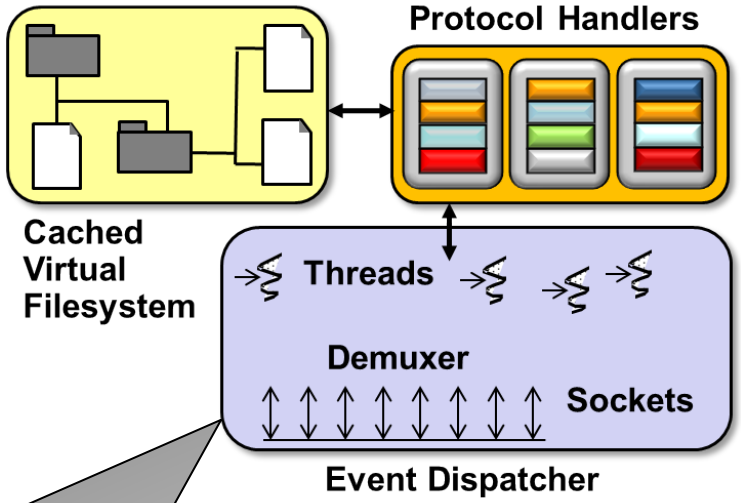
Professor of Computer Science

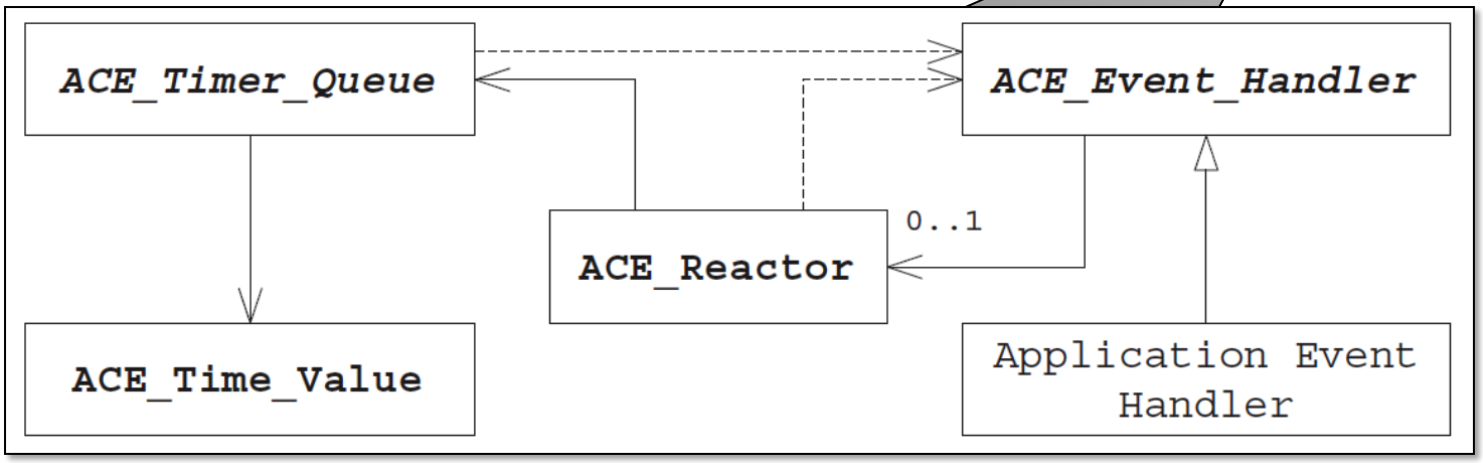
Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Topics Covered in this Part of the Module

- Describe the *Reactor* & *Acceptor-Connector* patterns
- Describe the ACE *Reactor* framework

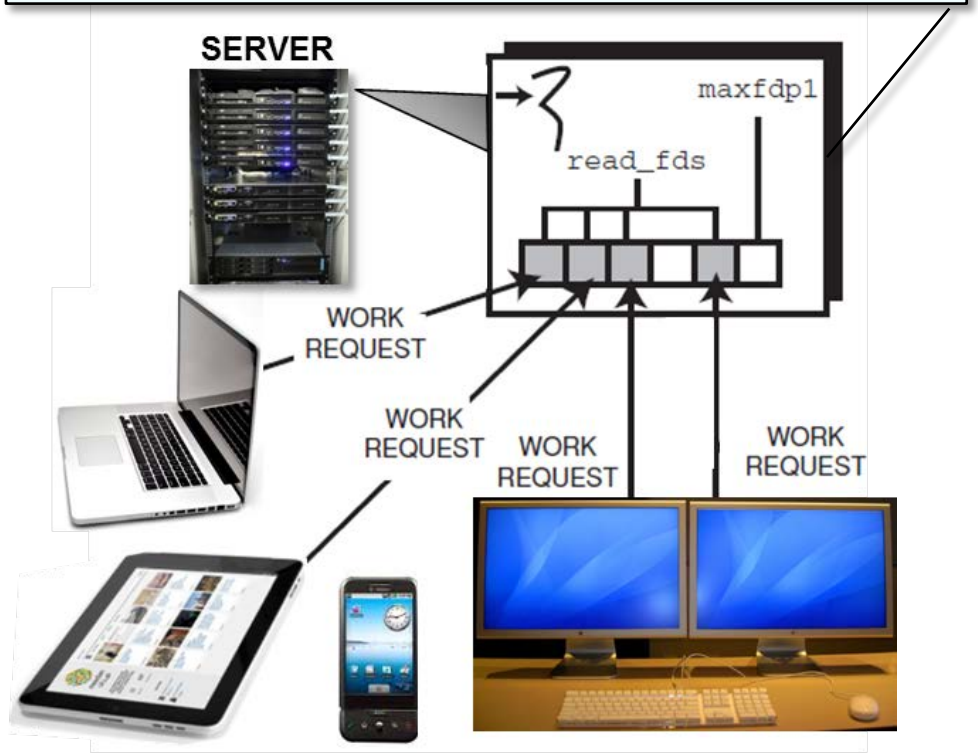




Motivation for the ACE Reactor Framework

- Event-driven networked app have historically been programmed via native OS mechanisms
  - e.g., Socket API & **`select()`**

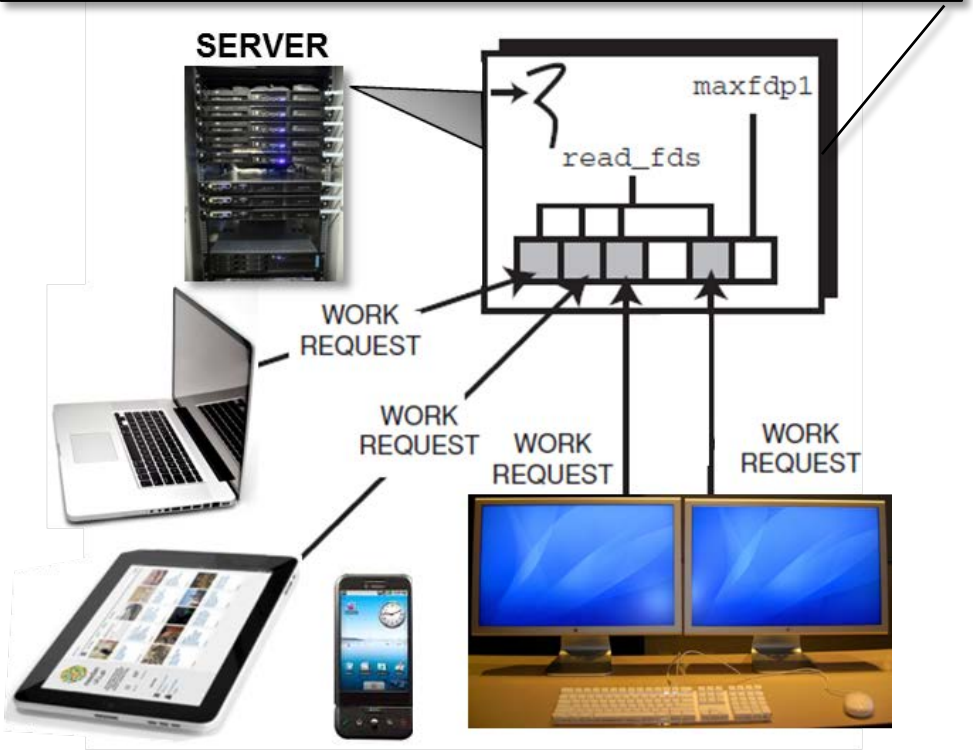
```
select (maxfdp1, &ready_handles, 0, 0, 0);
if (FD_ISSET (acceptor,  &ready_handles)) {
    int h = accept (acceptor, 0, 0);
    FD_SET (handle, &ready_handles);
    // ... Handle connection acceptance
} else
  for (int handle = s_handle + 1;
       handle < maxhandlep1; ++handle)
    if (FD_ISSET (handle, &ready_handles)) {
      // ... Perform app protocol processing
    }
```



Motivation for the ACE Reactor Framework

- Event-driven networked app have historically been programmed via native OS mechanisms
  - e.g., Socket API & `select()`
- Apps developed this way are often non-portable & inflexible
  - i.e., they tightly couple event detection, demuxing, & dispatching code together with app event processing code

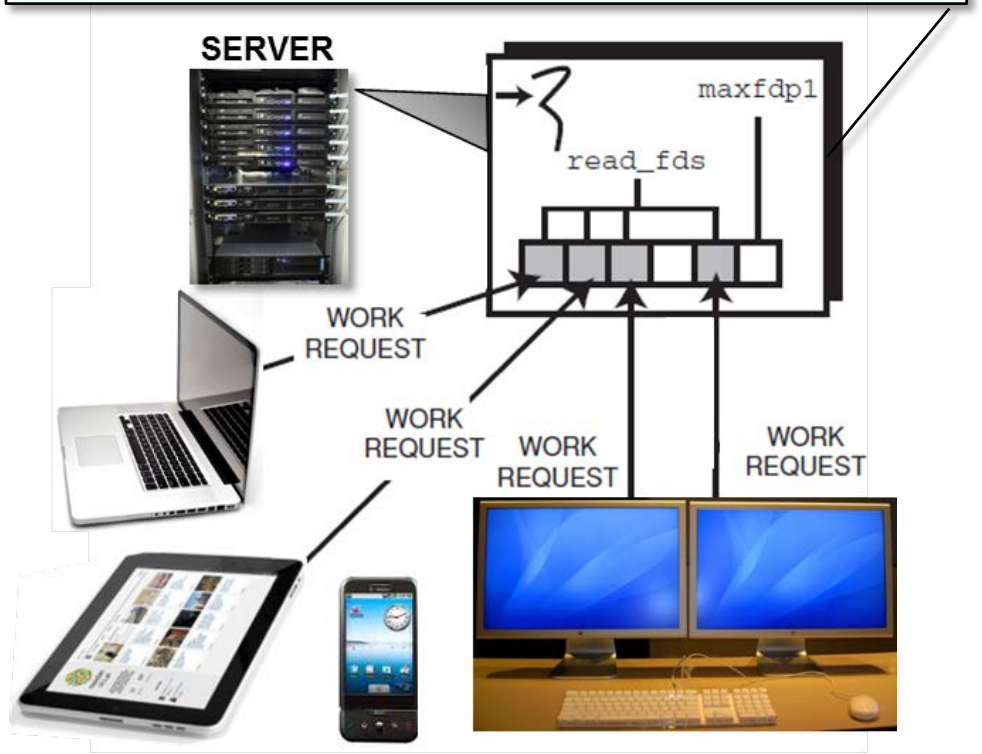
```
select (maxfdp1, &ready_handles, 0, 0, 0);
if (FD_ISSET (acceptor,  &ready_handles)) {
    int h = accept (acceptor, 0, 0);
    FD_SET (handle, &ready_handles);
    // ... Handle connection acceptance
} else
  for (int handle = s_handle + 1;
       handle < maxhandlep1; ++handle)
    if (FD_ISSET (handle, &ready_handles)) {
      // ... Perform app protocol processing
    }
```



Motivation for the ACE Reactor Framework

- Event-driven networked app have historically been programmed via native OS mechanisms
  - e.g., Socket API & **`select()`**
- Apps developed this way are often non-portable & inflexible
  - i.e., they tightly couple event detection, demuxing, & dispatching code together with app event processing code
- Developers therefore rewrite much code for each new app
  - Tedious, expensive, & error prone

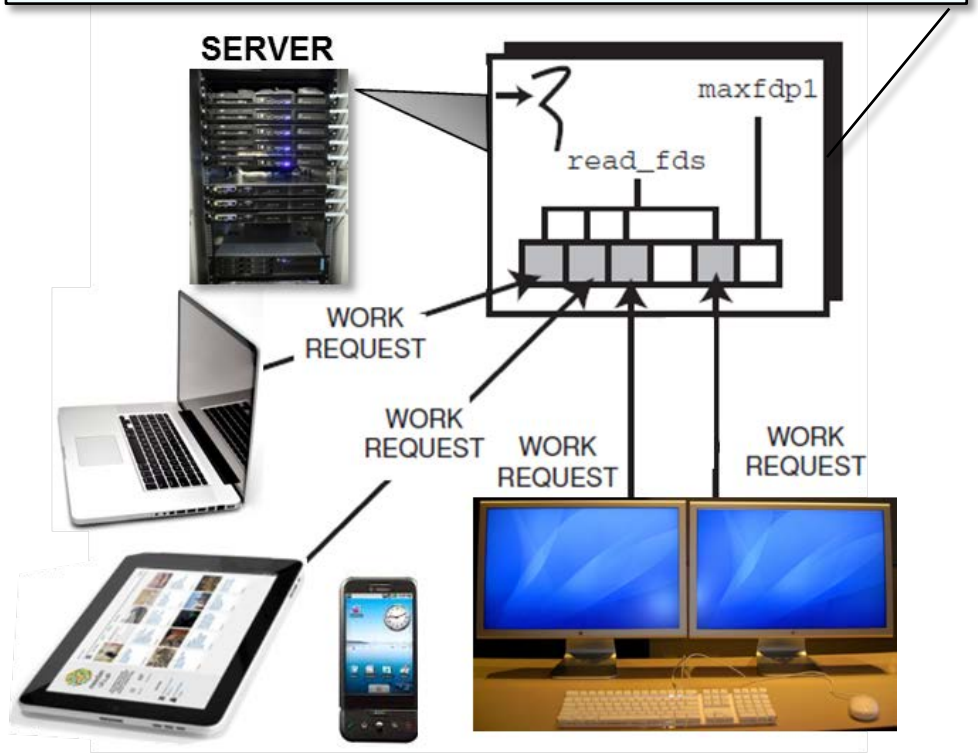
```
select (maxfdp1, &ready_handles, 0, 0, 0);
if (FD_ISSET (acceptor,  &ready_handles)) {
    int h = accept (acceptor, 0, 0);
    FD_SET (handle, &ready_handles);
    // ... Handle connection acceptance
} else
  for (int handle = s_handle + 1;
       handle < maxhandlep1; ++handle)
    if (FD_ISSET (handle, &ready_handles)) {
      // ... Perform app protocol processing
    }
```



Motivation for the ACE Reactor Framework

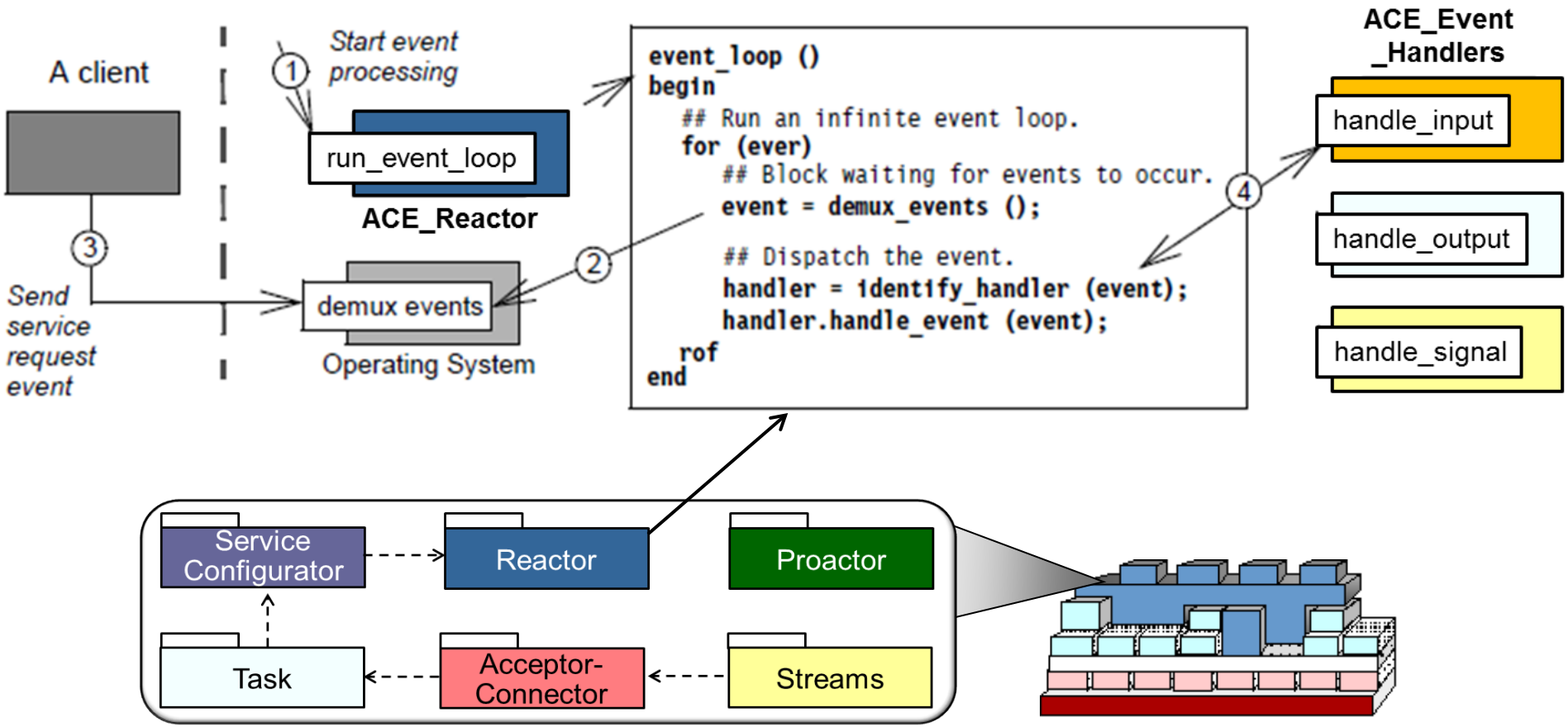
- Event-driven networked app have historically been programmed via native OS mechanisms
  - e.g., Socket API & **`select()`**
- Apps developed this way are often non-portable & inflexible
  - i.e., they tightly couple event detection, demuxing, & dispatching code together with app event processing code
- Developers therefore rewrite much code for each new app
  - Tedious, expensive, & error prone
- Much event detection, demuxing, & dispatching logic can be generalized & reused across many apps

```
select (maxfdp1, &ready_handles, 0, 0, 0);
if (FD_ISSET (acceptor,  &ready_handles)) {
    int h = accept (acceptor, 0, 0);
    FD_SET (handle, &ready_handles);
    // ... Handle connection acceptance
} else
  for (int handle = s_handle + 1;
       handle < maxhandlep1; ++handle)
    if (FD_ISSET (handle, &ready_handles)) {
      // ... Perform app protocol processing
    }
```



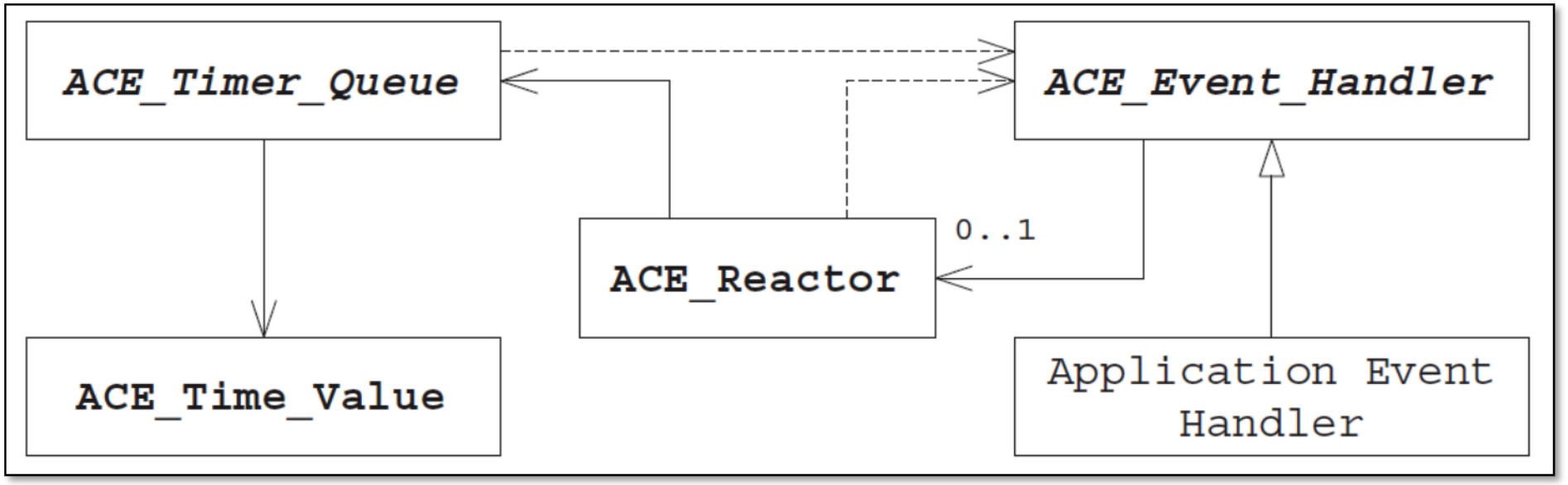
Overview of the ACE Reactor Framework

- Classes in this framework allow event-driven apps to react to events from a range of event sources
  - e.g., I/O handles, timers, & signals



Overview of the ACE Reactor Framework

- Classes in this framework allow event-driven apps to react to events from a range of event sources
- Apps inherit from **`ACE_Event_Handler`** & override its hook methods, which the ACE *Reactor* framework then dispatches to process events



These classes are designed in accordance with the *Reactor* pattern

Overview of the ACE Reactor Framework

- Classes in this framework allow event-driven apps to react to events from a range of event sources (e.g., I/O handles, timers, & signals)
- Apps inherit from **`ACE_Event_Handler`** & override its hook methods, which the ACE *Reactor* framework then dispatches to process events
- Key classes in the ACE *Reactor* framework include

ACE Class	Description
`ACE_Time_Value`	Provides a portable, normalized representation of time & duration that uses C++ operator overloading to simplify time-related arithmetic & relational operations
`ACE_Timer_Queue`	An abstract class defining the capabilities & interface for a timer queue (ACE provides several subclasses that provide flexible support for various timing requirements)
`ACE_Event_Handler`	An abstract class whose interface defines the hook methods that are the targets of **`ACE_Reactor`** callbacks
`ACE_Reactor`	Provides the interface for managing event handler registrations & executing the event loop that drives event detection, demultiplexing, & dispatching in the ACE *Reactor* framework

Overview of the ACE Reactor Framework

- Classes in this framework allow event-driven apps to react to events from a range of event sources (e.g., I/O handles, timers, & signals)
- Apps inherit from **`ACE_Event_Handler`** & override its hook methods, which the ACE *Reactor* framework then dispatches to process events
- Key classes in the ACE *Reactor* framework include

ACE Class	Description
`ACE_Time_Value`	Provides a portable, normalized representation of time & duration that uses C++ operator overloading to simplify time-related arithmetic & relational operations
`ACE_Timer_Queue`	An abstract class defining the capabilities & interface for a timer queue (ACE provides several subclasses that provide flexible support for various timing requirements)
`ACE_Event_Handler`	An abstract class whose interface defines the hook methods that are the targets of **`ACE_Reactor`** callbacks
`ACE_Reactor`	Provides the interface for managing event handler registrations & executing the event loop that drives event detection, demultiplexing, & dispatching in the ACE *Reactor* framework

Overview of the ACE Reactor Framework

- Classes in this framework allow event-driven apps to react to events from a range of event sources (e.g., I/O handles, timers, & signals)
- Apps inherit from **`ACE_Event_Handler`** & override its hook methods, which the ACE *Reactor* framework then dispatches to process events
- Key classes in the ACE *Reactor* framework include

ACE Class	Description
`ACE_Time_Value`	Provides a portable, normalized representation of time & duration that uses C++ operator overloading to simplify time-related arithmetic & relational operations
`ACE_Timer_Queue`	An abstract class defining the capabilities & interface for a timer queue (ACE provides several subclasses that provide flexible support for various timing requirements)
`ACE_Event_Handler`	An abstract class whose interface defines the hook methods that are the targets of **`ACE_Reactor`** callbacks
`ACE_Reactor`	Provides the interface for managing event handler registrations & executing the event loop that drives event detection, demultiplexing, & dispatching in the ACE *Reactor* framework

Overview of the ACE Reactor Framework

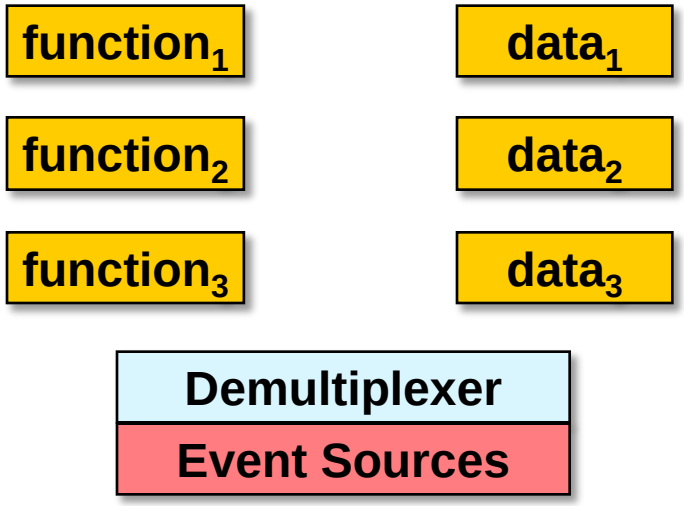
- Classes in this framework allow event-driven apps to react to events from a range of event sources (e.g., I/O handles, timers, & signals)
- Apps inherit from **`ACE_Event_Handler`** & override its hook methods, which the ACE *Reactor* framework then dispatches to process events
- Key classes in the ACE *Reactor* framework include

ACE Class	Description
`ACE_Time_Value`	Provides a portable, normalized representation of time & duration that uses C++ operator overloading to simplify time-related arithmetic & relational operations
`ACE_Timer_Queue`	An abstract class defining the capabilities & interface for a timer queue (ACE provides several subclasses that provide flexible support for various timing requirements)
`ACE_Event_Handler`	An abstract class whose interface defines the hook methods that are the targets of **`ACE_Reactor`** callbacks
`ACE_Reactor`	Provides the interface for managing event handler registrations & executing the event loop that drives event detection, demultiplexing, & dispatching in the ACE *Reactor* framework

See www.dre.vanderbilt.edu/~schmidt/PDF/reactor-rules.pdf for more info

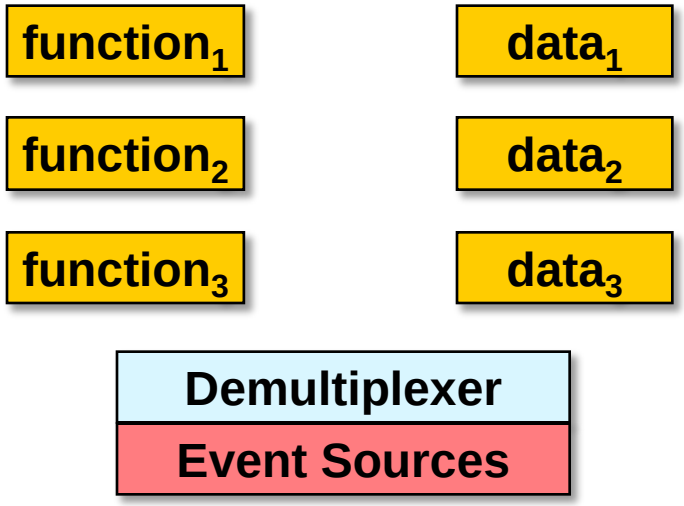
Motivation for Object-Oriented Event Handling

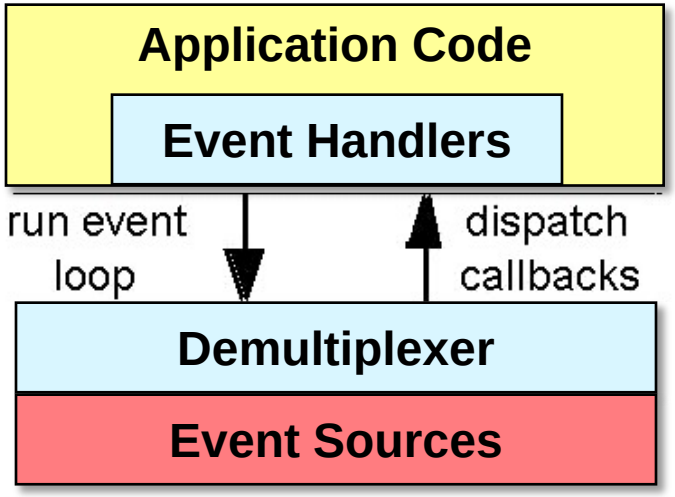
- Networked apps are often event driven & their processing is driven by callbacks to event handlers
- Implementing callbacks by defining a separate function handlers for each type of event lacks cohesion & is error-prone



Motivation for Object-Oriented Event Handling

- Networked apps are often event driven & their processing is driven by callbacks to event handlers
- Implementing callbacks by defining a separate function handlers for each type of event lacks cohesion & is error-prone



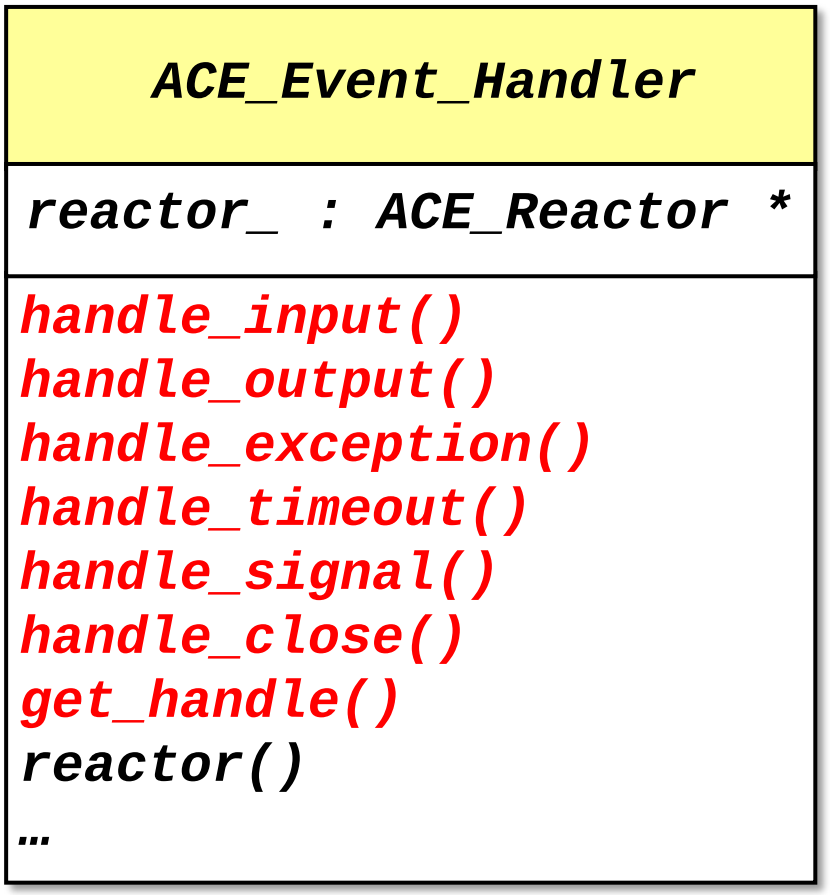


- It is more effective to devise a more cohesive *object-oriented* event demuxing mechanism
  - e.g., implement callbacks via event handler objects that *integrate* data & methods

The ACE_Event_Handler Class

Base class of all reactive event handlers provides the following capabilities:

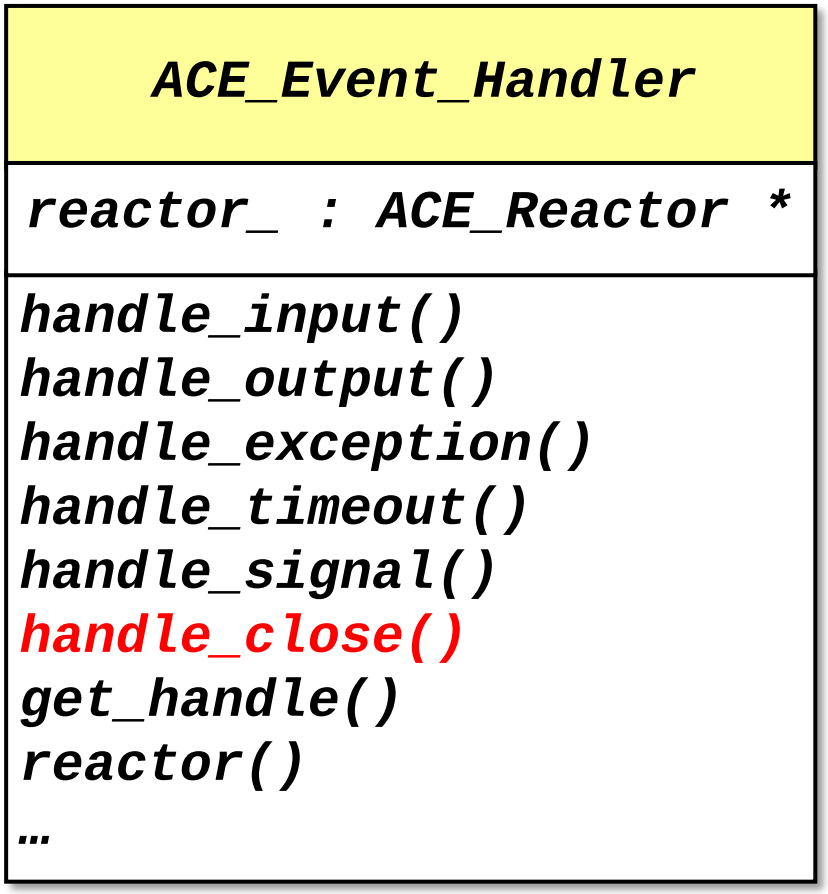
- Hook methods for input, output, exception, timer, & signal events



The ACE_Event_Handler Class

Base class of all reactive event handlers provides the following capabilities:

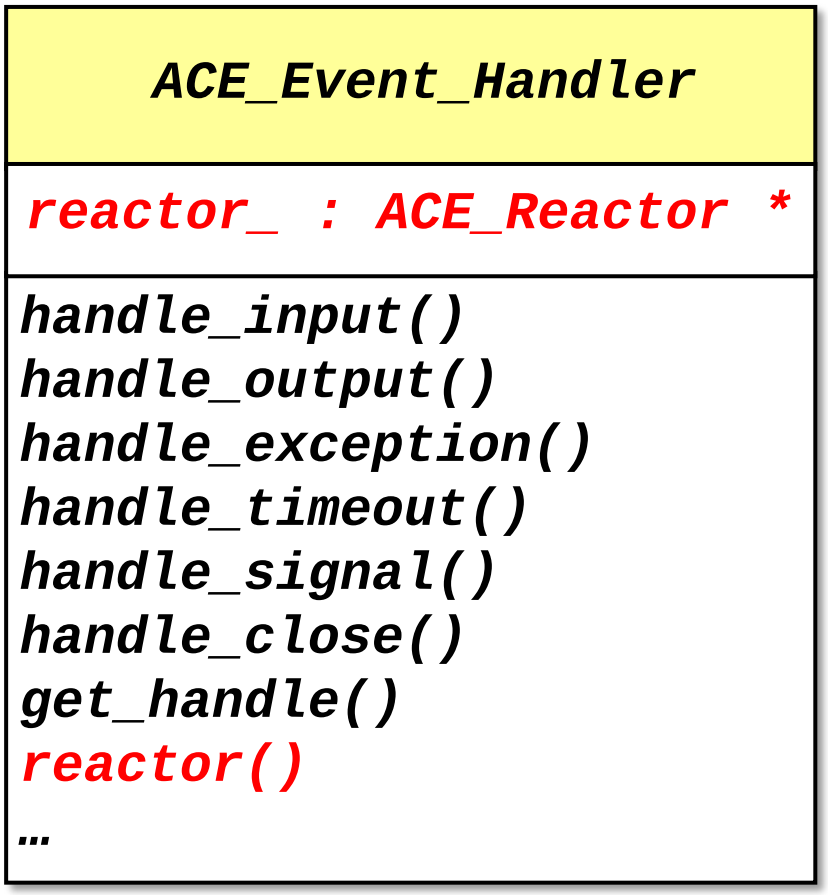
- Hook methods for input, output, exception, timer, & signal events
- It centralizes how event handlers are destroyed when they're not needed



The ACE_Event_Handler Class

Base class of all reactive event handlers provides the following capabilities:

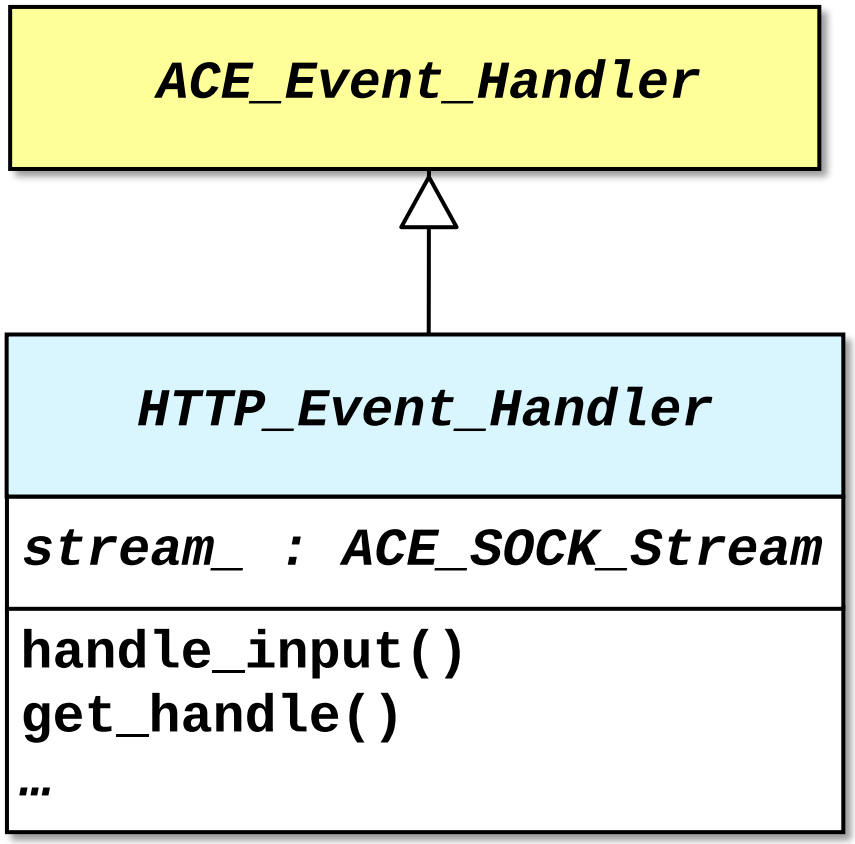
- Hook methods for input, output, exception, timer, & signal events
- It centralizes how event handlers are destroyed when they're not needed
- It holds a pointer to an instance of the **`ACE_Reactor`** that manages it
  - Allows an event handler to manage its event (de)registration



The ACE_Event_Handler Class

Base class of all reactive event handlers provides the following capabilities:

- Hook methods for input, output, exception, timer, & signal events
- It centralizes how event handlers are destroyed when they're not needed
- It holds a pointer to an instance of the **`ACE_Reactor`** that manages it
  - Allows an event handler to manage its event (de)registration
- OO callbacks simplify association of data with hook methods that manipulate data



Handles *variability* of event processing via a *common* event handler API

Types of Events & Event Handler Hooks

- When an app registers an event handler with a reactor, it must indicate what type(s) of event(s) the event handler should process

```
int HTTP_Event_Handler::open ()
{
  return reactor ()->register_handler
           (this, ACE_Event_Handler::READ_MASK);
}
```

Register with the reactor to receive input events containing GET requests

Types of Events & Event Handler Hooks

- When an app registers an event handler with a reactor, it must indicate what type(s) of event(s) the event handler should process
- ACE designates these event types via enumerators defined in **`ACE_Event_Handler`** that are associated with the **`handle_*()`** hook methods

Event Type	Description
`READ_MASK`	Indicates input events, such as data on a socket or file handle – a reactor dispatches **`handle input()`** to process input events
`WRITE_MASK`	Indicates output events, such as when flow control abates – a reactor dispatches the **`handle output()`** to process output events
`EXCEPT_MASK`	Indicates exceptional events, such as urgent data on a socket – a reactor dispatches **`handle exception()`** to process exceptional events
`ACCEPT_MASK`	Indicates passive-mode connection events – a reactor dispatches the **`handle input()`** to process connection events
`CONNECT_MASK`	Indicates a non-blocking connection completion – a reactor dispatches **`handle output()`** to process non-blocking connection completion events

Types of Events & Event Handler Hooks

- When an app registers an event handler with a reactor, it must indicate what type(s) of event(s) the event handler should process
- ACE designates these event types via enumerators defined in **`ACE_Event_Handler`** that are associated with the **`handle_*()`** hook methods

Event Type	Description
`READ_MASK`	Indicates input events, such as data on a socket or file handle – a reactor dispatches **`handle input()`** to process input events
`WRITE_MASK`	Indicates output events, such as when flow control abates – a reactor dispatches the **`handle output()`** to process output events
`EXCEPT_MASK`	Indicates exceptional events, such as urgent data on a socket – a reactor dispatches **`handle exception()`** to process exceptional events
`ACCEPT_MASK`	Indicates passive-mode connection events – a reactor dispatches the **`handle input()`** to process connection events
`CONNECT_MASK`	Indicates a non-blocking connection completion – a reactor dispatches **`handle output()`** to process non-blocking connection completion events

- These values can be combined ("or'd" together) to designate a set of events
  - These events can populate the **`ACE_Reactor_Mask`** parameter passed to **`ACE_Reactor::register_handler()`**

Event Handler Hook Method Return Values

- When registered events occur, the reactor dispatches the appropriate event handler's **`handle_*()`** hook methods to process them

Called back by reactor when a data event occurs to memory-map requested file & send its contents back to the client

```
int HTTP_Event_Handler::handle_input (ACE_HANDLE) {
  std::string pathname (get_pathname (stream_));

  ACE_Mem_Map mapped_file (pathname.c_str ());

  peer ().send_n (mapped_file.addr (),
                  mapped_file.size ());

  return ...
}
```

See next slide for summary of return values

Event Handler Hook Method Return Values

- When registered events occur, the reactor dispatches the appropriate event handler's **`handle_*()`** hook methods to process them
- When a **`handle_*()`** method finishes its processing, it must return a value that's interpreted by the reactor as follows:

Returns	Behavior
Zero (0)	• Indicates reactor should continue to detect & dispatch registered events for this handler/handle
Minus one (−1)	• Instructs the reactor to stop detecting registered event for this handler/handle & call its **`handle_close()`** hook method
Greater than zero (> 0)	• Indicates that the reactor should dispatch this event handler on the handle again before it blocks on its event multiplexer (at which point it will continue to detect & dispatch registered events for this handler/handle)

Event Handler Hook Method Return Values

- When registered events occur, the reactor dispatches the appropriate event handler's **`handle_*()`** hook methods to process them
- When a **`handle_*()`** method finishes its processing, it must return a value that's interpreted by the reactor as follows:

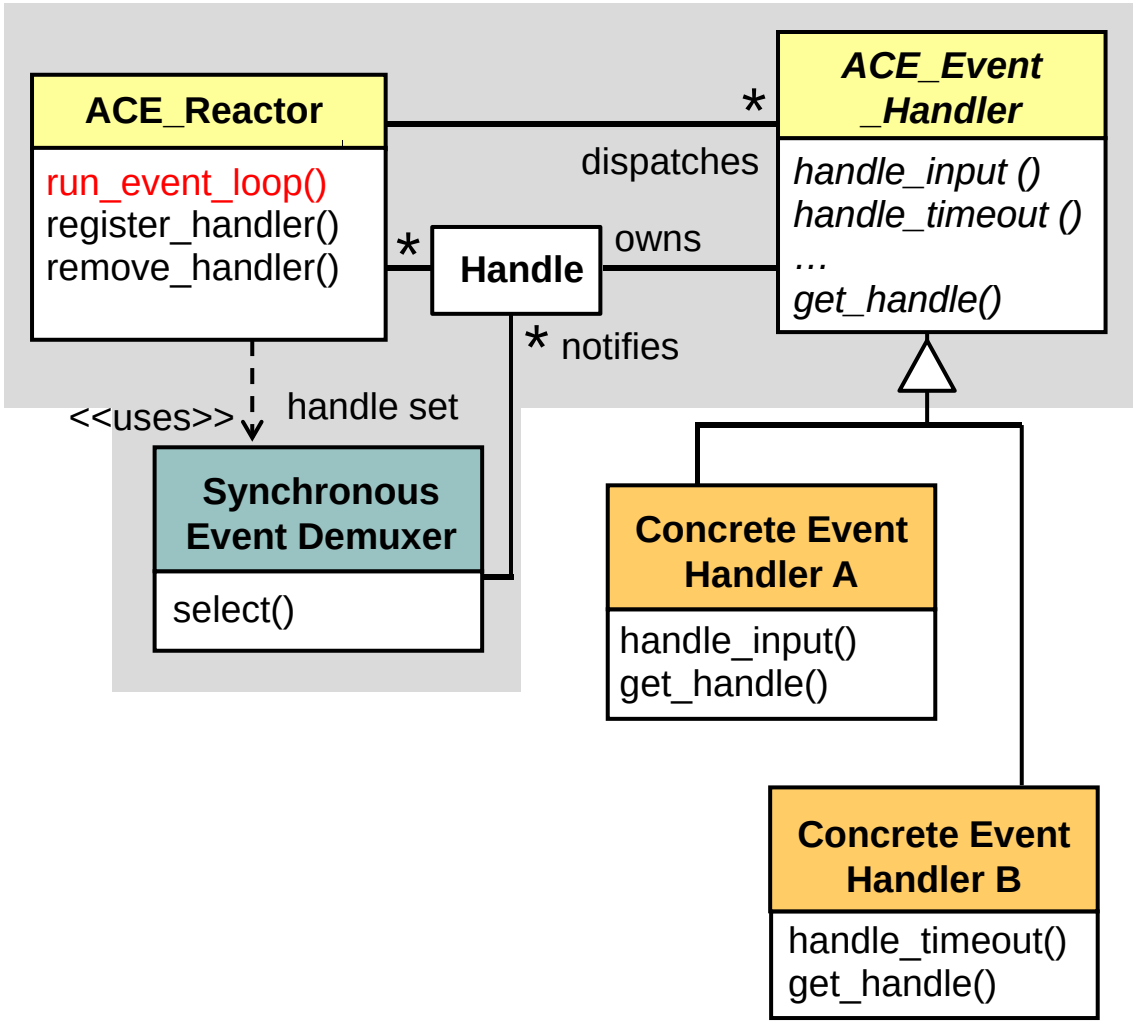
Returns	Behavior
Zero (0)	• Indicates reactor should continue to detect & dispatch registered events for this handler/handle
Minus one (−1)	• Instructs the reactor to stop detecting registered event for this handler/handle & call its **`handle_close()`** hook method
Greater than zero (> 0)	• Indicates that the reactor should dispatch this event handler on the handle again before it blocks on its event multiplexer (at which point it will continue to detect & dispatch registered events for this handler/handle)

- Before the reactor removes an event handler, it invokes the handler's hook method **`handle_close()`**, passing **`ACE_Reactor_Mask`** of the event that's being unregistered

The ACE_Reactor Class

Defines an interface for ACE *Reactor* framework capabilities:

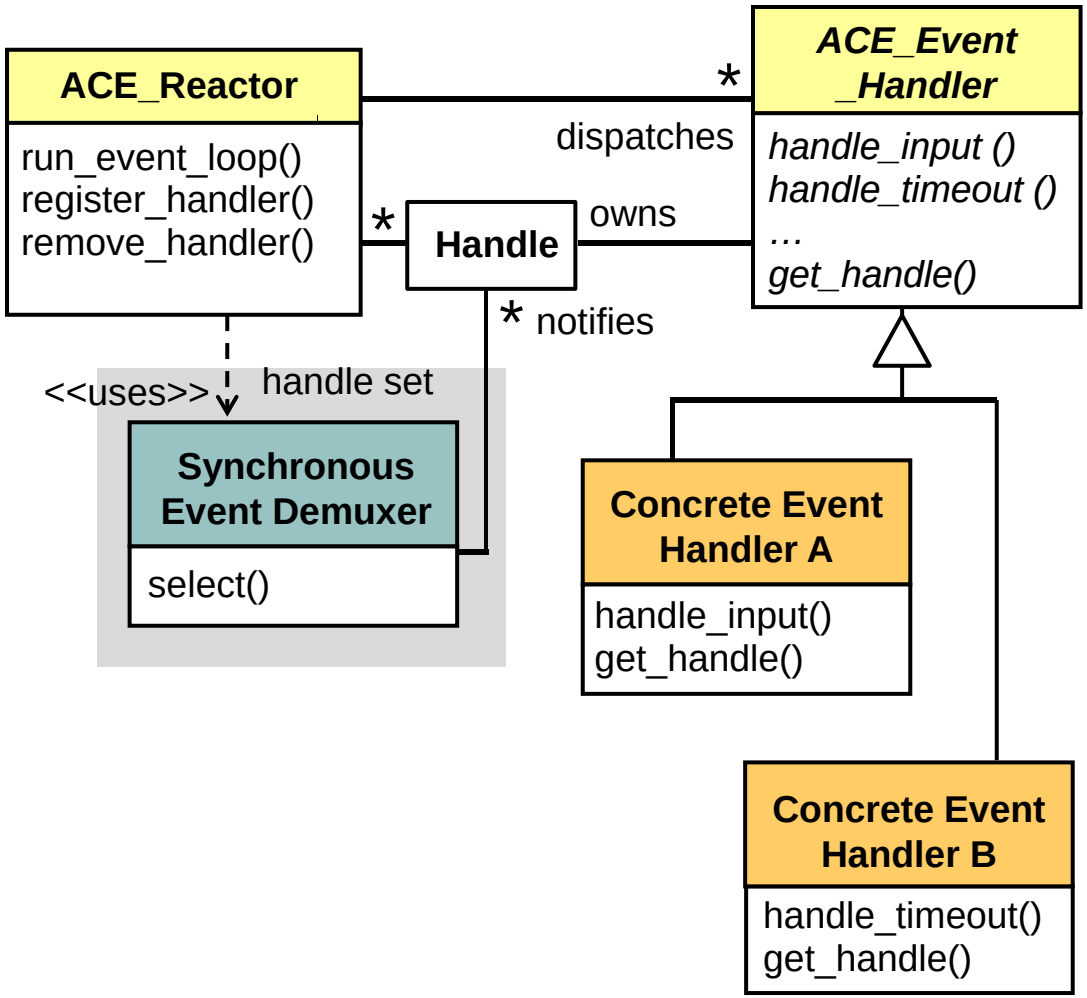
- Centralizes event loop processing



The ACE_Reactor Class

Defines an interface for ACE *Reactor* framework capabilities:

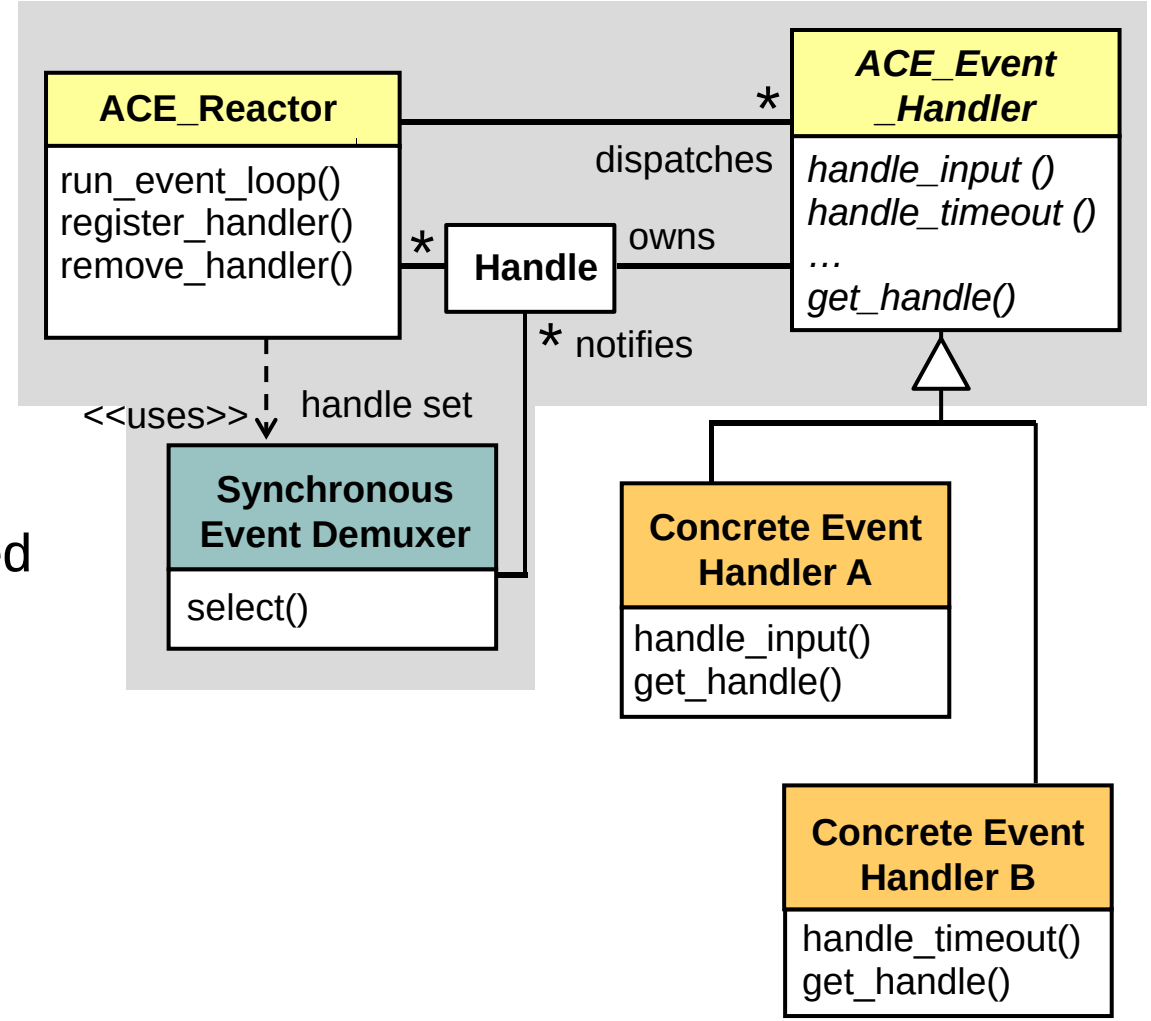
- Centralizes event loop processing
- Detects events via an OS event muxer



The ACE_Reactor Class

Defines an interface for ACE *Reactor* framework capabilities:

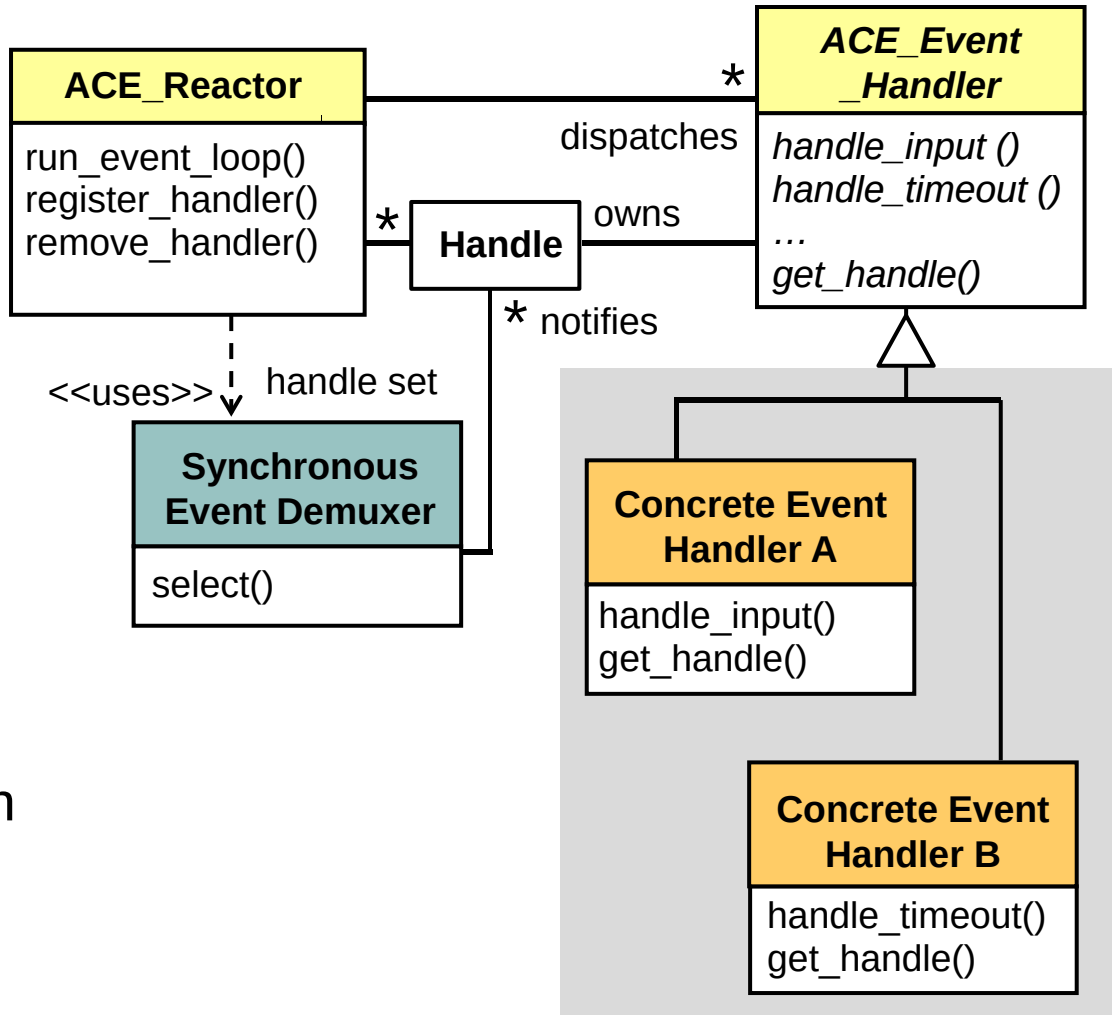
- Centralizes event loop processing
- Detects events via an OS event muxer
- Demuxes events to registered event handlers when event muxer detects occurrence of events



The ACE_Reactor Class

Defines an interface for ACE *Reactor* framework capabilities:

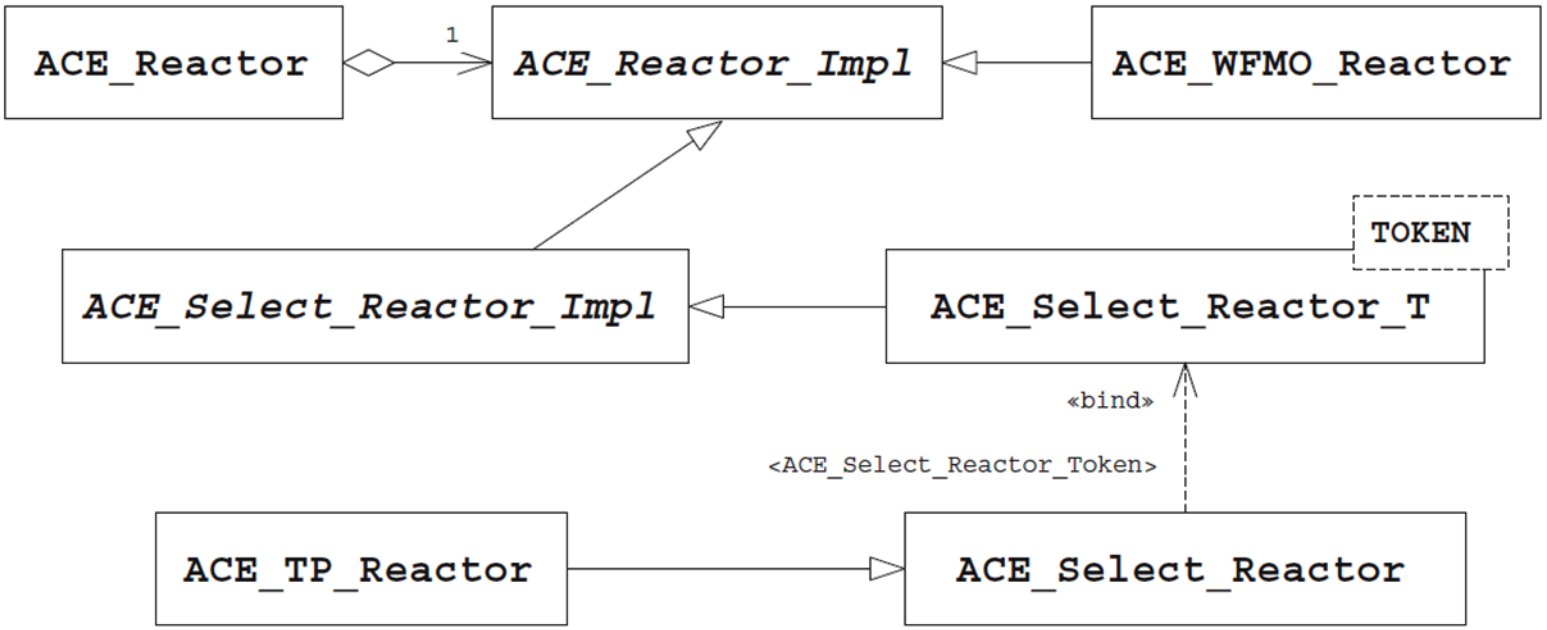
- Centralizes event loop processing
- Detects events via an OS event muxer
- Demuxes events to registered event handlers when event muxer detects occurrence of events
- Dispatches hook method(s) on event handler(s) to perform app-defined processing in response to events



Handles *variability* of synchronous event (de)muxing via a *common* API

ACE Reactor Implementations

- The ACE *Reactor* framework was designed for extensibility
  - Internally it uses the *Bridge* pattern & there are nearly a dozen reactor implementations in ACE



ACE Reactor Implementations

- The ACE *Reactor* framework was designed for extensibility
  - Internally it uses the *Bridge* pattern & there are nearly a dozen reactor implementations in ACE
- The most common Reactor implementations include:

ACE Class	Description
`ACE_Select_Reactor`	Uses the **`select()`** event muxer function to detect I/O & timer events; incorporates orderly handling of POSIX signals
`ACE_TP_Reactor`	Uses the Leader/Followers pattern to extend **`ACE_Select_Reactor`** event handling to a pool of threads
`ACE_WFMO_Reactor`	Uses the Windows **`WaitForMultipleObjects()`** event muxer function to detect socket I/O, timeouts, & Windows events
`ACE_Dev_Poll_Reactor`	Uses the UNIX **`/dev/poll`** & **`/dev/epoll`** event muxer functions that are more scalable than **`select()`**

ACE Reactor Implementations

- The ACE *Reactor* framework was designed for extensibility
  - Internally it uses the *Bridge* pattern & there are nearly a dozen reactor implementations in ACE
- The most common Reactor implementations include:

ACE Class	Description
`ACE_Select_Reactor`	Uses the **`select()`** event muxer function to detect I/O & timer events; incorporates orderly handling of POSIX signals
`ACE_TP_Reactor`	Uses the Leader/Followers pattern to extend **`ACE_Select_Reactor`** event handling to a pool of threads
`ACE_WFMO_Reactor`	Uses the Windows **`WaitForMultipleObjects()`** event muxer function to detect socket I/O, timeouts, & Windows events
`ACE_Dev_Poll_Reactor`	Uses the UNIX **`/dev/poll`** & **`/dev/epoll`** event muxer functions that are more scalable than **`select()`**

ACE Reactor Implementations

- The ACE *Reactor* framework was designed for extensibility
  - Internally it uses the *Bridge* pattern & there are nearly a dozen reactor implementations in ACE
- The most common Reactor implementations include:

ACE Class	Description
`ACE_Select_Reactor`	Uses the **`select()`** event muxer function to detect I/O & timer events; incorporates orderly handling of POSIX signals
`ACE_TP_Reactor`	Uses the Leader/Followers pattern to extend **`ACE_Select_Reactor`** event handling to a pool of threads
`ACE_WFMO_Reactor`	Uses the Windows **`WaitForMultipleObjects()`** event muxer function to detect socket I/O, timeouts, & Windows events
`ACE_Dev_Poll_Reactor`	Uses the UNIX **`/dev/poll`** & **`/dev/epoll`** event muxer functions that are more scalable than **`select()`**

ACE Reactor Implementations

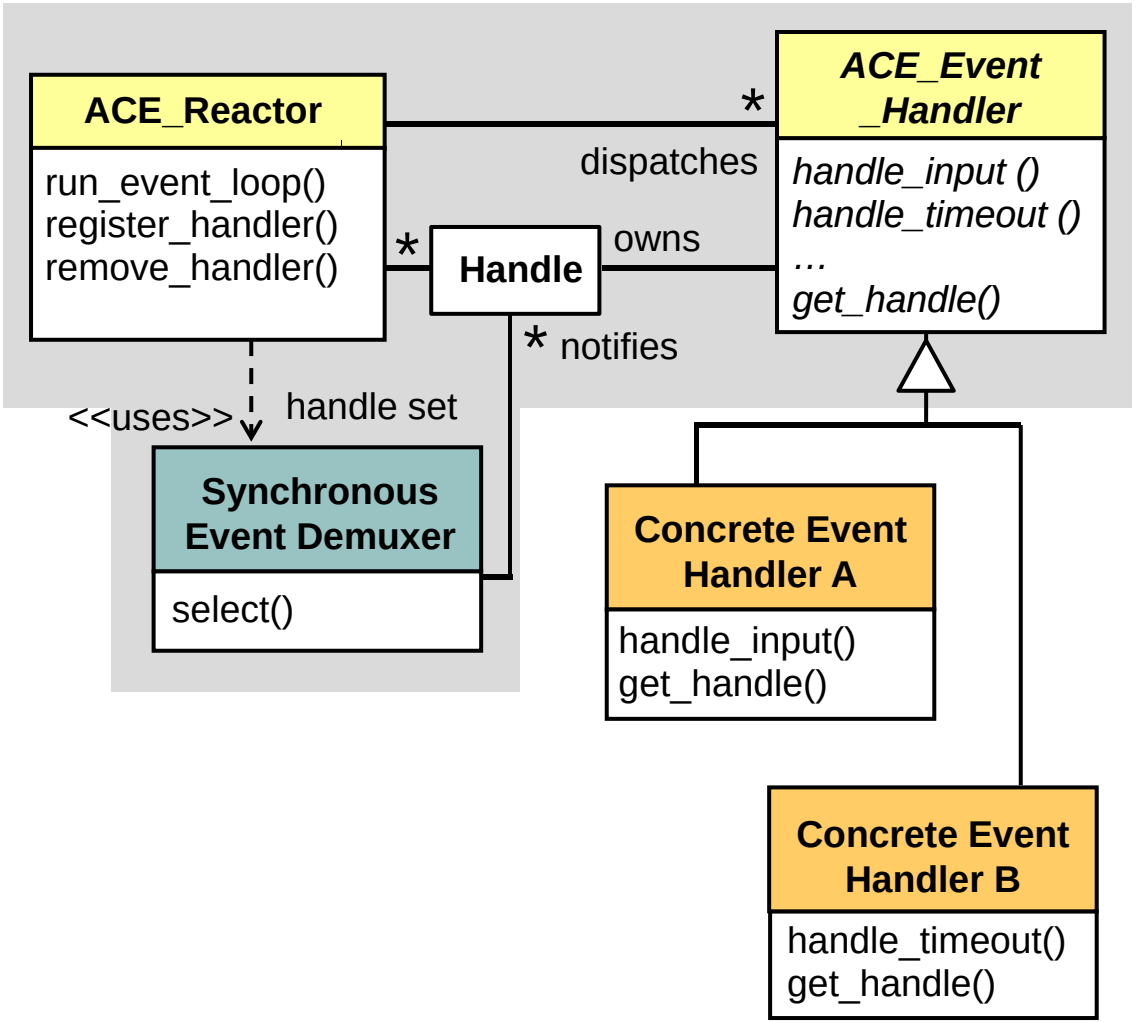
- The ACE *Reactor* framework was designed for extensibility
  - Internally it uses the *Bridge* pattern & there are nearly a dozen reactor implementations in ACE
- The most common Reactor implementations include:

ACE Class	Description
`ACE_Select_Reactor`	Uses the **`select()`** event muxer function to detect I/O & timer events; incorporates orderly handling of POSIX signals
`ACE_TP_Reactor`	Uses the Leader/Followers pattern to extend **`ACE_Select_Reactor`** event handling to a pool of threads
`ACE_WFMO_Reactor`	Uses the Windows **`WaitForMultipleObjects()`** event muxer function to detect socket I/O, timeouts, & Windows events
`ACE_Dev_Poll_Reactor`	Uses the Solaris **`/dev/poll`** & Linux **`/dev/epoll`** event muxer functions that are more scalable than **`select()`**

Summary

The ACE *Reactor* framework

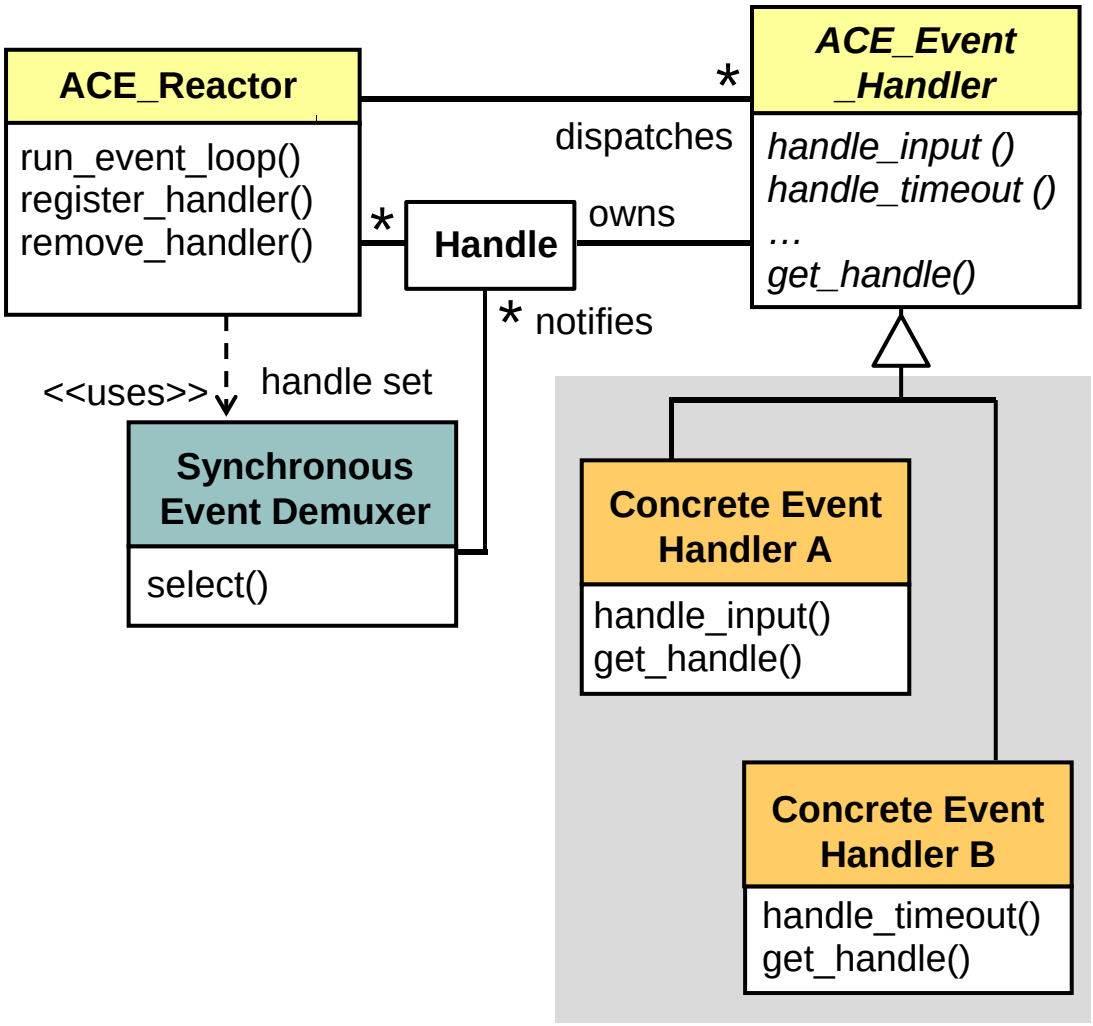
- Simplifies the development of concise, correct, portable, & efficient event-driven networked apps
  - Encapsulates OS event muxing mechanisms within an OO C++ interface



Summary

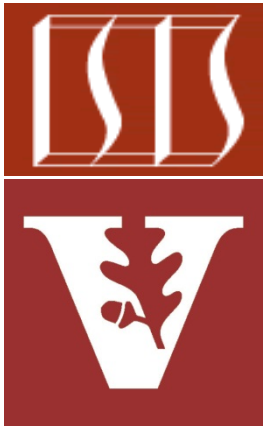
The ACE *Reactor* framework

- Simplifies the development of concise, correct, portable, & efficient event-driven networked apps
  - Encapsulates OS event muxing mechanisms within an OO C++ interface
- Enhances reuse, improves portability, & enables the extensibility of event handlers
  - Separates generic event detection, demuxing, & dispatching *mechanisms* from app-defined event processing *policies*



Patterns & Frameworks for Synchronous Event Handling, Connections, & Service Initialization: Part 3

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt



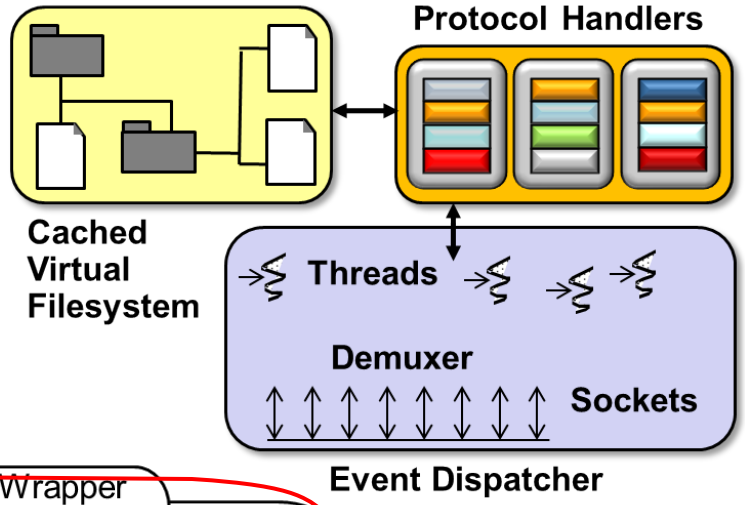
Professor of Computer Science

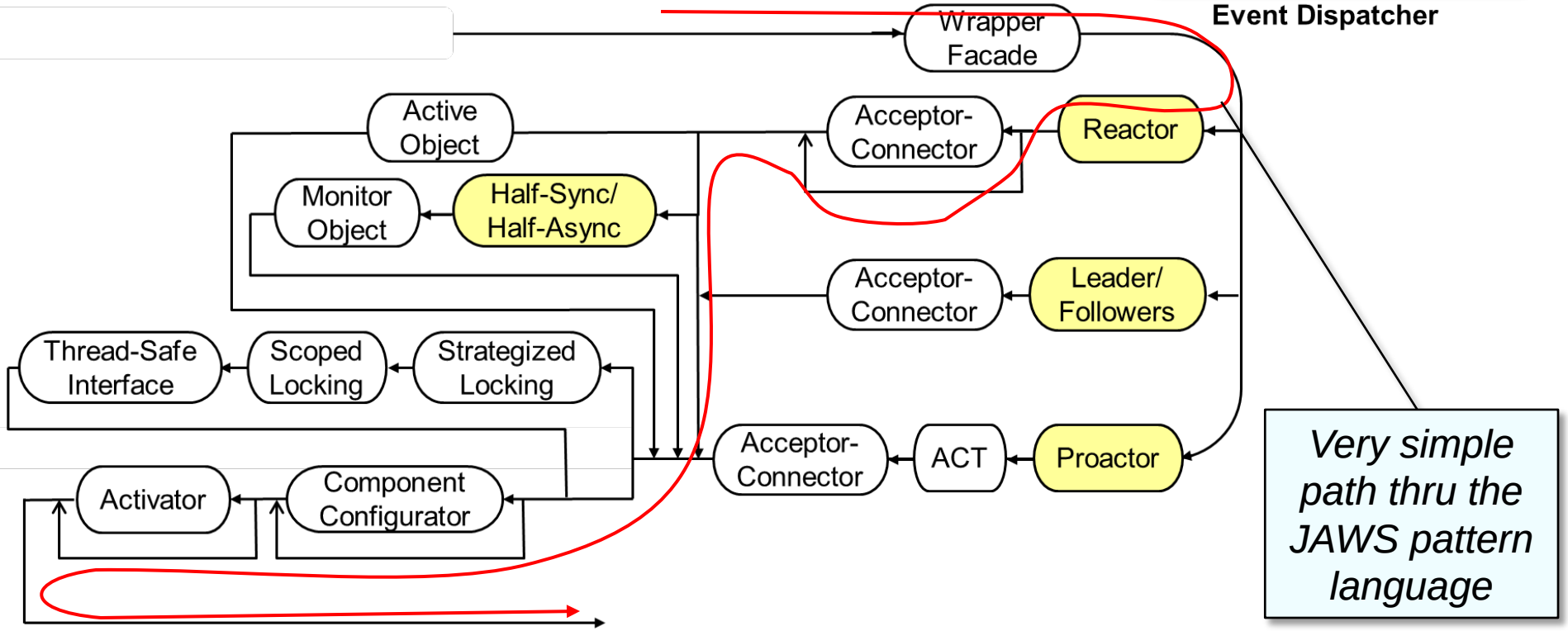
Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

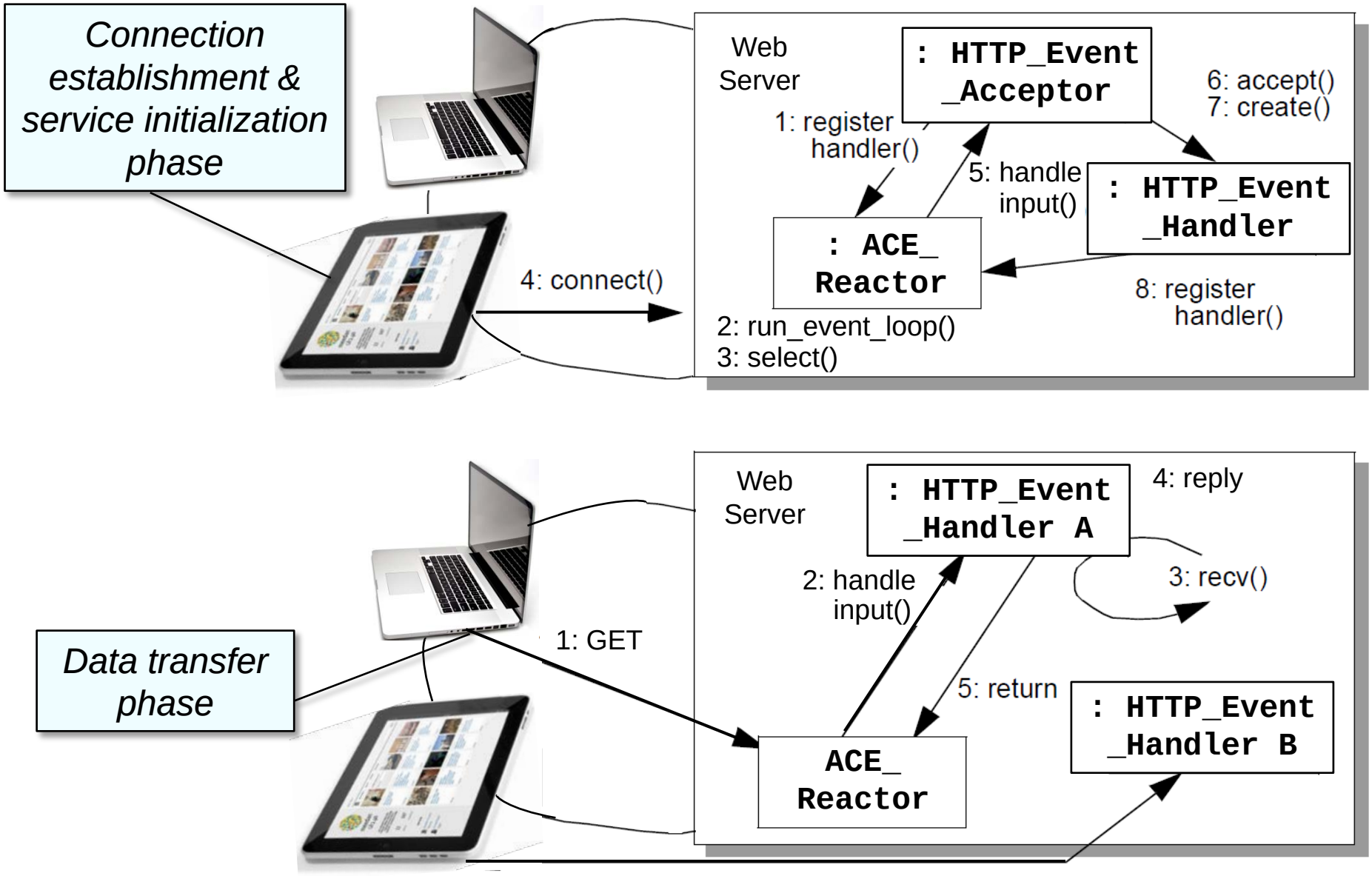
Topics Covered in this Part of the Module

- Describe the *Reactor* & *Acceptor-Connector* patterns
- Describe the ACE *Reactor* framework
- Apply ACE *Reactor* framework to JAWS



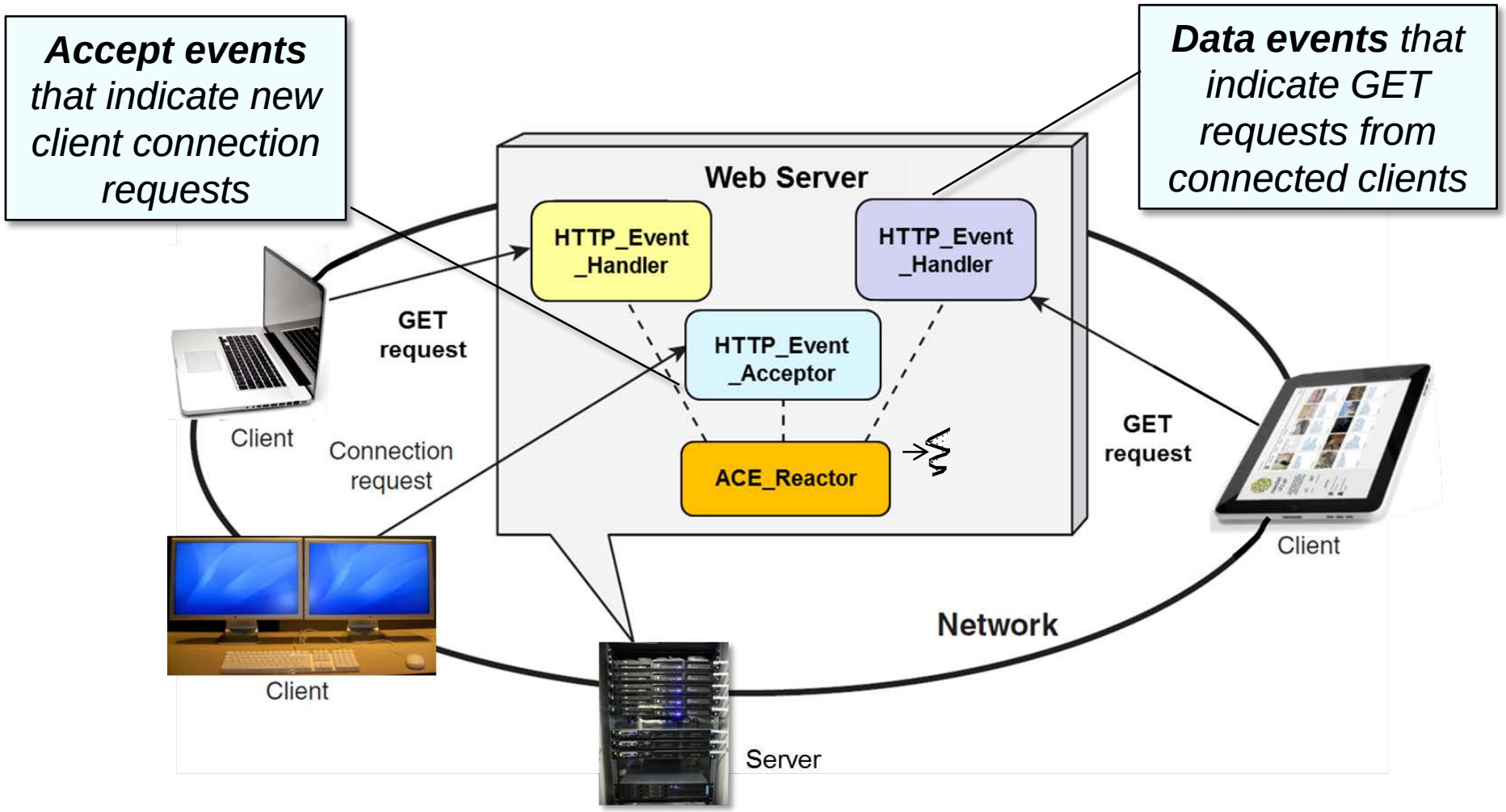


ACE Reactor Implementation of JAWS



Using the ACE_Event_Handler Class with JAWS

We implement a reactive JAWS web server by inheriting from **`ACE_Event_Handler`** & driving its processing via the reactor's event loop to handle two types of events:



Using ACE_Event_Handler with JAWS

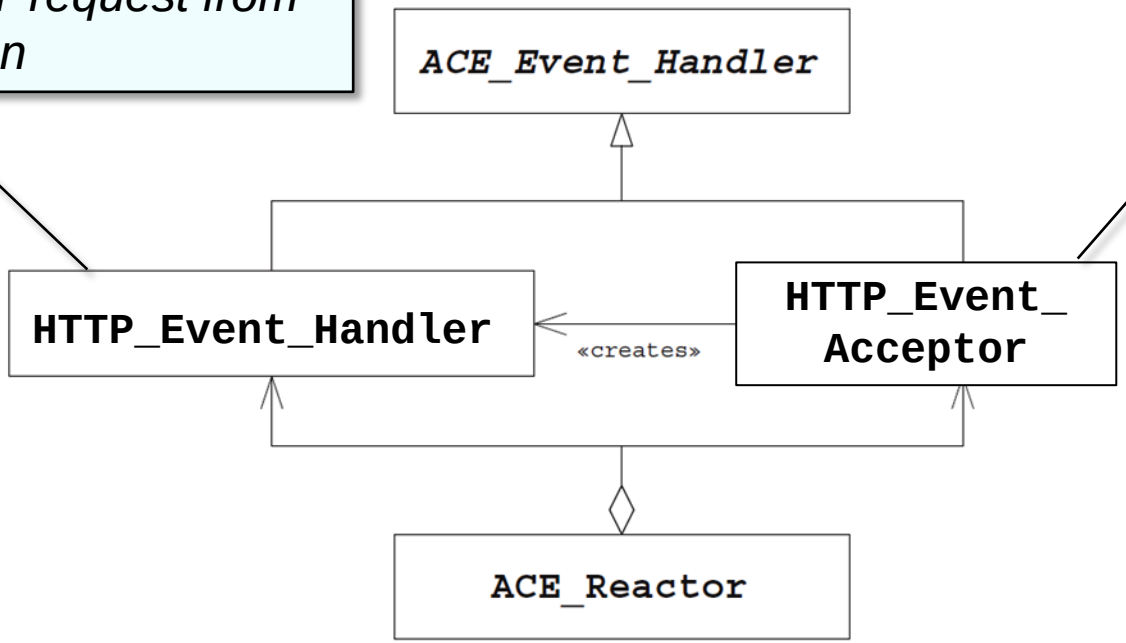
We define two types of event handlers in our reactive web server:

HTTP_Event_Handler

- *Processes GET requests received from connected clients*
- *Uses* ***ACE_SOCK_Stream*** *to read GET request from a connection*

HTTP_Event_Acceptor

- *Factory that allocates* ***HTTP_Event_Handler*** *dynamically & initializes it when client connects*
- *Uses* ***ACE_SOCK_Acceptor*** *to initialize* ***ACE_SOCK_Stream*** *contained in* ***HTTP_Event_Handler***



Using ACE_Event_Handler with JAWS

- The **HTTP_Event_Acceptor** is a factory that allocates an **HTTP_Event_Handler** dynamically & initializes it when a client connects to the web server

```
class HTTP_Event_Acceptor : public ACE_Event_Handler {
private:
  ACE_SOCK_Acceptor acceptor_;

public:
  HTTP_Event_Acceptor (ACE_Reactor *r = ACE_Reactor::instance())
    : ACE_Event_Handler (r) {}

  virtual int open (const ACE_INET_Addr &local_addr) {
    acceptor_.open (local_addr);
    return reactor ()->register_handler
           (this, ACE_Event_Handler::ACCEPT_MASK);
}
```

Factory that passively connects ACE_SOCK_Streams

Note default use of reactor singleton

Initialization forwards to ACE_SOCK_Acceptor

Register this object with the reactor for accept events

Some error checking has been omitted to reduce clutter

Using ACE_Event_Handler with JAWS

Hook method called when object removed from reactor

```
virtual int handle_close (ACE_HANDLE, ACE_Reactor_Mask) {
  acceptor_.close ();
  delete this;
  return 0;
}
```

It's ok to "delete this" when the event handler is destroyed!

```
virtual ACE_HANDLE get_handle () const
{ return acceptor_.get_handle (); }
```

Return passive-mode socket's I/O handle

```
virtual int handle_input (ACE_HANDLE);
};
```

Key hook method dispatched by reactor when there's a new connection to accept (implementation shown shortly)

Using ACE_Event_Handler with JAWS

- The **HTTP_Event_Handler** processes GET request from connected client

```
class HTTP_Event_Handler : public ACE_Event_Handler {
protected:
   ACE_SOCK_Stream stream_;

public:
  HTTP_Event_Handler (ACE_Reactor *r): ACE_Event_Handler (r) {}

  virtual int open ();

  ACE_SOCK_Stream &peer () { return stream_; };

  virtual int handle_input (ACE_HANDLE);

  virtual int handle_close (ACE_HANDLE, ACE_Reactor_Mask);
};
```

Connection to remote peer

Activation hook method

Get reference to contained ACE_SOCK_Stream

Key hook method dispatched by reactor when GET requests arrive

Called by a reactor when handler is closing

Using ACE_Event_Handler with JAWS

Factory method called back by reactor when a connection event occurs

```
1 int HTTP_Event_Acceptor::handle_input (ACE_HANDLE) {
2   HTTP_Event_Handler *peer_handler = new (peer_handler,
3                                           HTTP_Event_Handler (reactor ()));
4   acceptor_.accept (peer_handler->peer ());
    ...
5   peer_handler->open ();
    ...
```

Create event handler to receive GET requests

Accept connection & store in ACE_SOCK_STREAM

Activate connected HTTP_Event_Handler

Using ACE_Event_Handler with JAWS

```
1 int HTTP_Event_Acceptor::handle_input (ACE_HANDLE) {
2   HTTP_Event_Handler *peer_handler = new (peer_handler,
3                                 HTTP_Event_Handler (reactor ()));

4   acceptor_.accept (peer_handler->peer ());


   ...
5  peer_handler->open ();
   ...
```

```
int HTTP_Event_Handler::open () {
  return reactor ()->register_handler
           (this, ACE_Event_Handler::READ_MASK);
}
```

Register with the reactor to receive input events containing GET requests

Using ACE_Event_Handler with JAWS

Called back by the reactor when a data event occurs

```
int HTTP_Event_Handler::handle_input (ACE_HANDLE) {
  std::string pathname (get_pathname (stream_));
  ACE_Mem_Map mapped_file (pathname.c_str ());

  if (peer ().send_n (mapped_file.addr (),
                      mapped_file.size ()) == -1) return -1;
  peer ().close ();
  return 0;
}

int HTTP_Event_Handler::handle_close (ACE_HANDLE,
                                      ACE_Reactor_Mask) {
  stream_.close ();
  delete this;
  return 0;
}
```

Memory map the requested content

Transmits content back to client

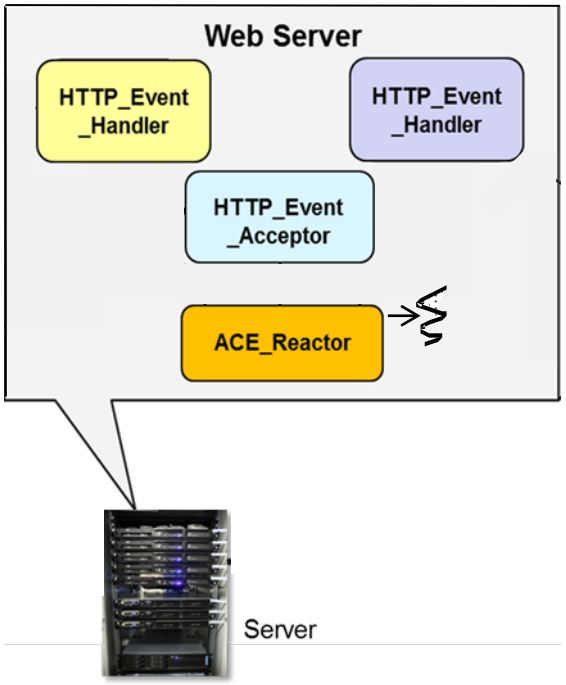
Called back by the reactor when `handle_input()` returns `-1`

Note similarity of **`handle_input()`** logic wrt earlier *Wrapper Façade* solution

Using ACE_Reactor with JAWS

This example illustrates a web server that

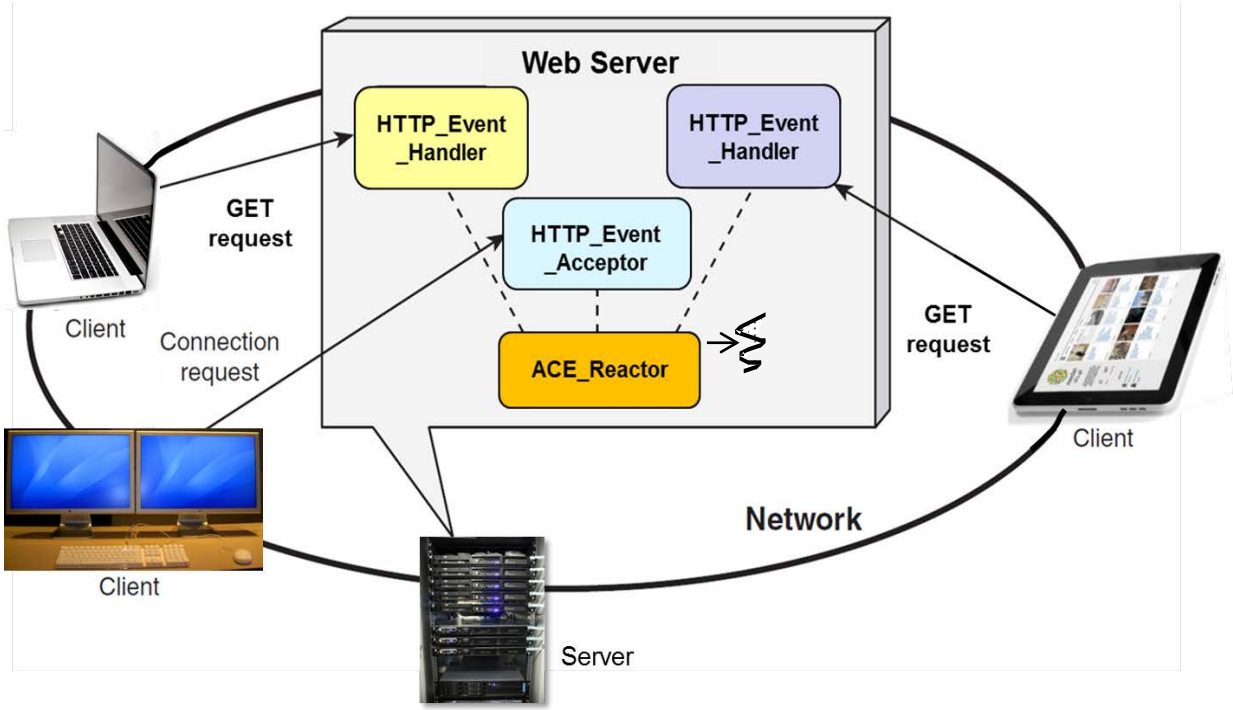
- Runs in a single thread of control in a single process



Using ACE_Reactor with JAWS

This example illustrates a web server that

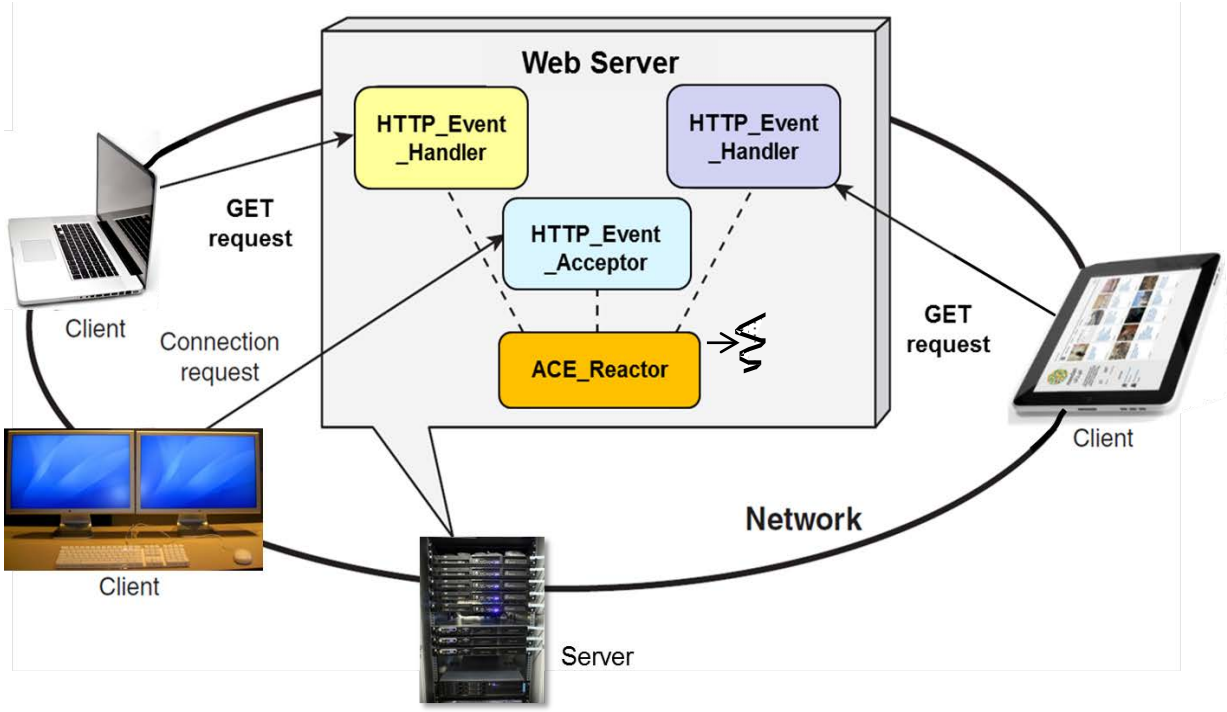
- Runs in a single thread of control in a single process
- Handles connection & GET requests from multiple clients reactively



Using ACE_Reactor with JAWS

This example illustrates a web server that

- Runs in a single thread of control in a single process
- Handles connection & GET requests from multiple clients reactively



Template parameter is used as base class

```
template <typename ACCEPTOR>
class Reactor_HTTP_Server : public ACCEPTOR {
public:
  Reactor_HTTP_Server (int argc, char *argv[], ACE_Reactor *);
};
```

Using ACE_Reactor with JAWS

```
1 template <typename ACCEPTOR>
2 Reactor_HTTP_Server<ACCEPTOR>::Reactor_HTTP_Server
3   (int argc, char *argv[], ACE_Reactor *reactor)
4   : ACCEPTOR (reactor) {
5   u_short HTTP_port = argc > 1 ? atoi (argv[1]) : 80;

6   typename ACCEPTOR::PEER_ADDR server_addr
                                  (HTTP_port, INADDR_ANY);
7   if (ACCEPTOR::open (addr) == -1)
8     reactor->end_reactor_event_loop ();
9 }
```

Initialize ACCEPTOR base class with reactor pointer

Begin listening for connections

Socket address to listen on for connections

Shutdown reactor's event loop if an error occurs

Using ACE_Reactor with JAWS

```
1 typedef Reactor_HTTP_Server<HTTP_Event_Acceptor>
2         HTTP_Server_Daemon;
3 int main (int argc, char *argv[]) {
4   ACE_Reactor reactor;
5   new HTTP_Server_Daemon (argc, argv, &reactor);
6
7   reactor.run_reactor_event_loop ()
    ...
```

Instantiate template with HTTP_Event_Acceptor

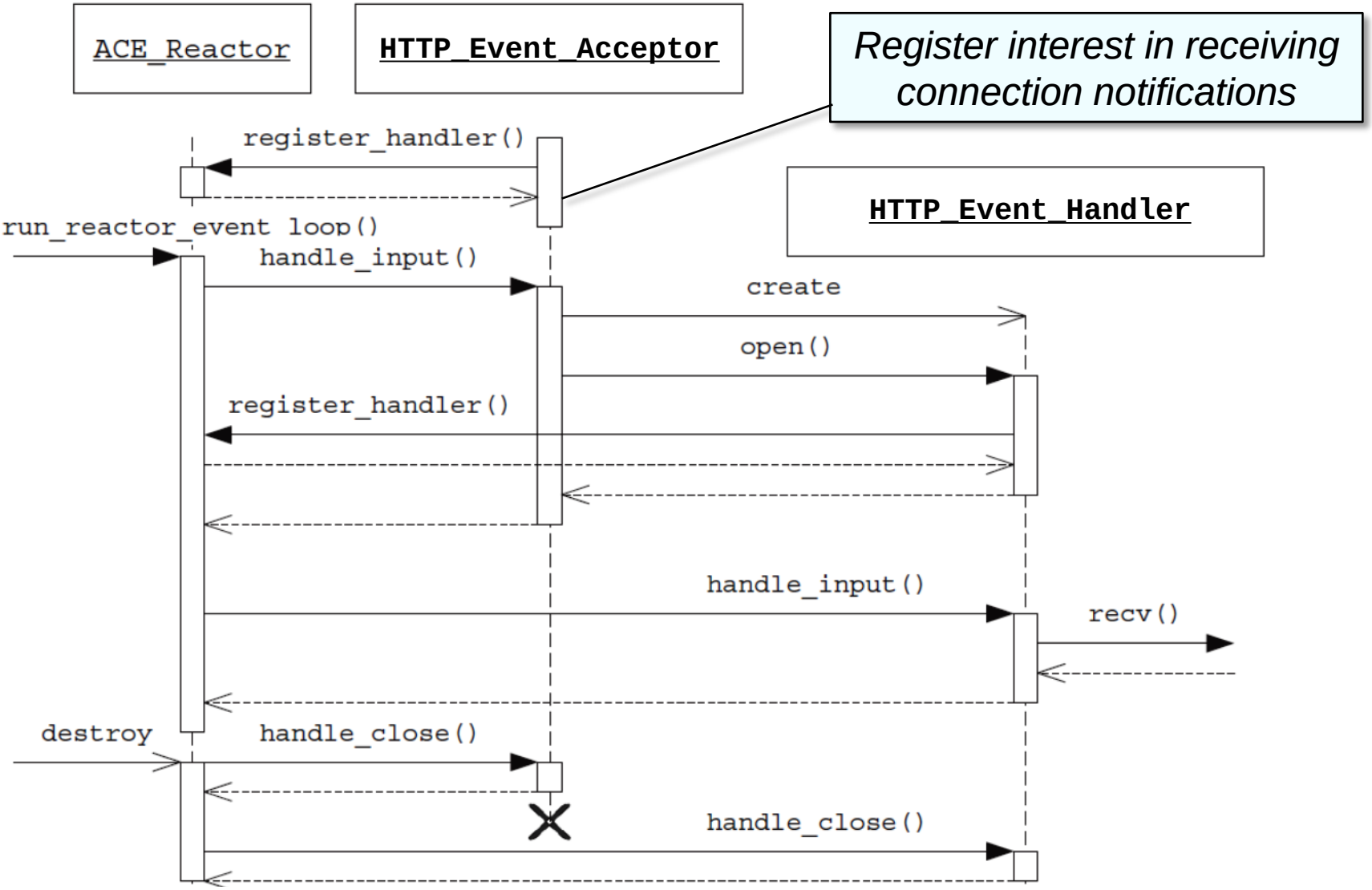
Event loop controller object

Dynamic allocation ensures proper deletion semantics

Run web server event loop

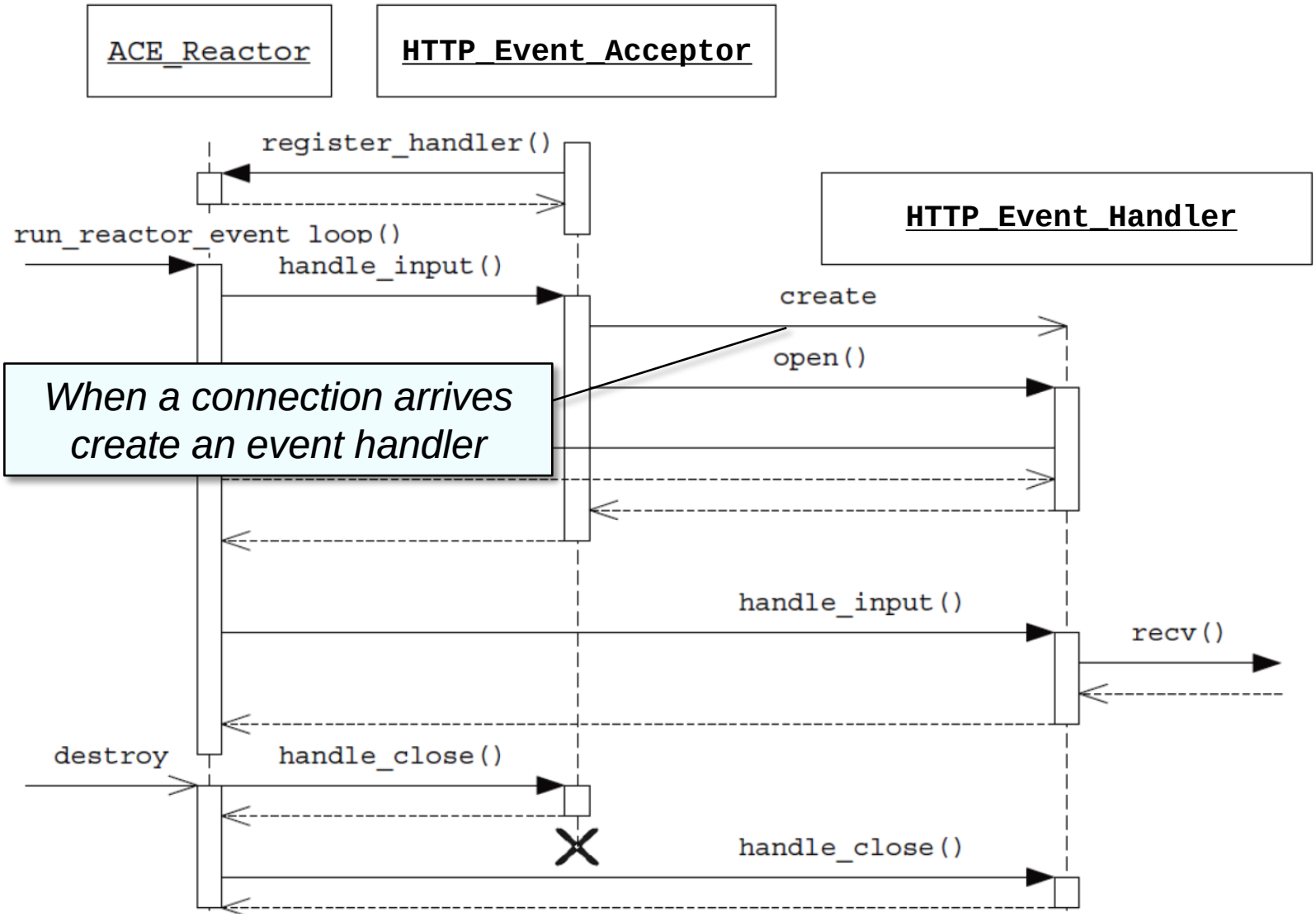
This reactive event loop structure can be further generalized, as we'll see later

Using ACE_Reactor with JAWS



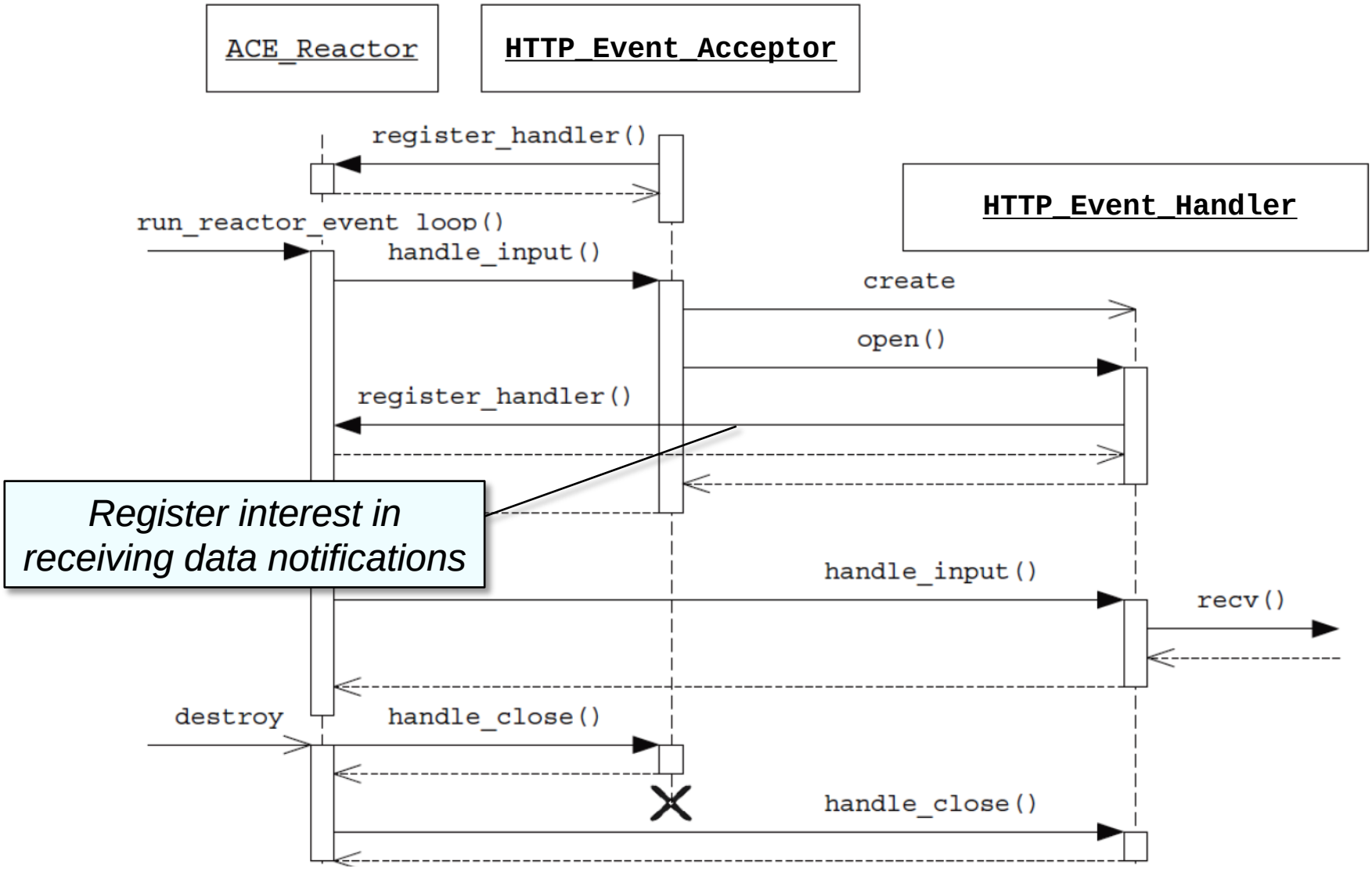
Sequence diagram for ACE *Reactor*-based web server

Using ACE_Reactor with JAWS



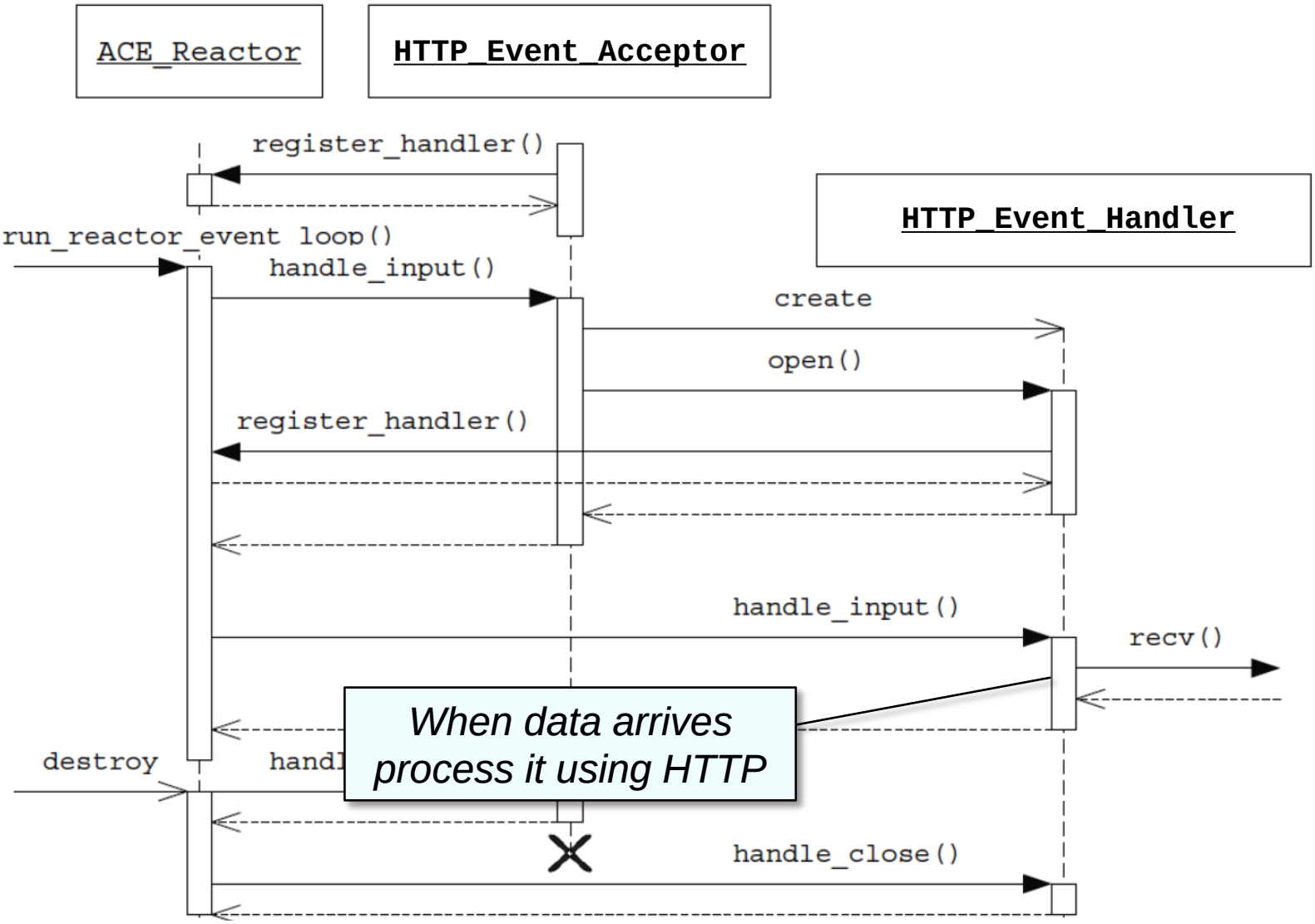
Sequence diagram for ACE *Reactor*-based web server

Using ACE_Reactor with JAWS



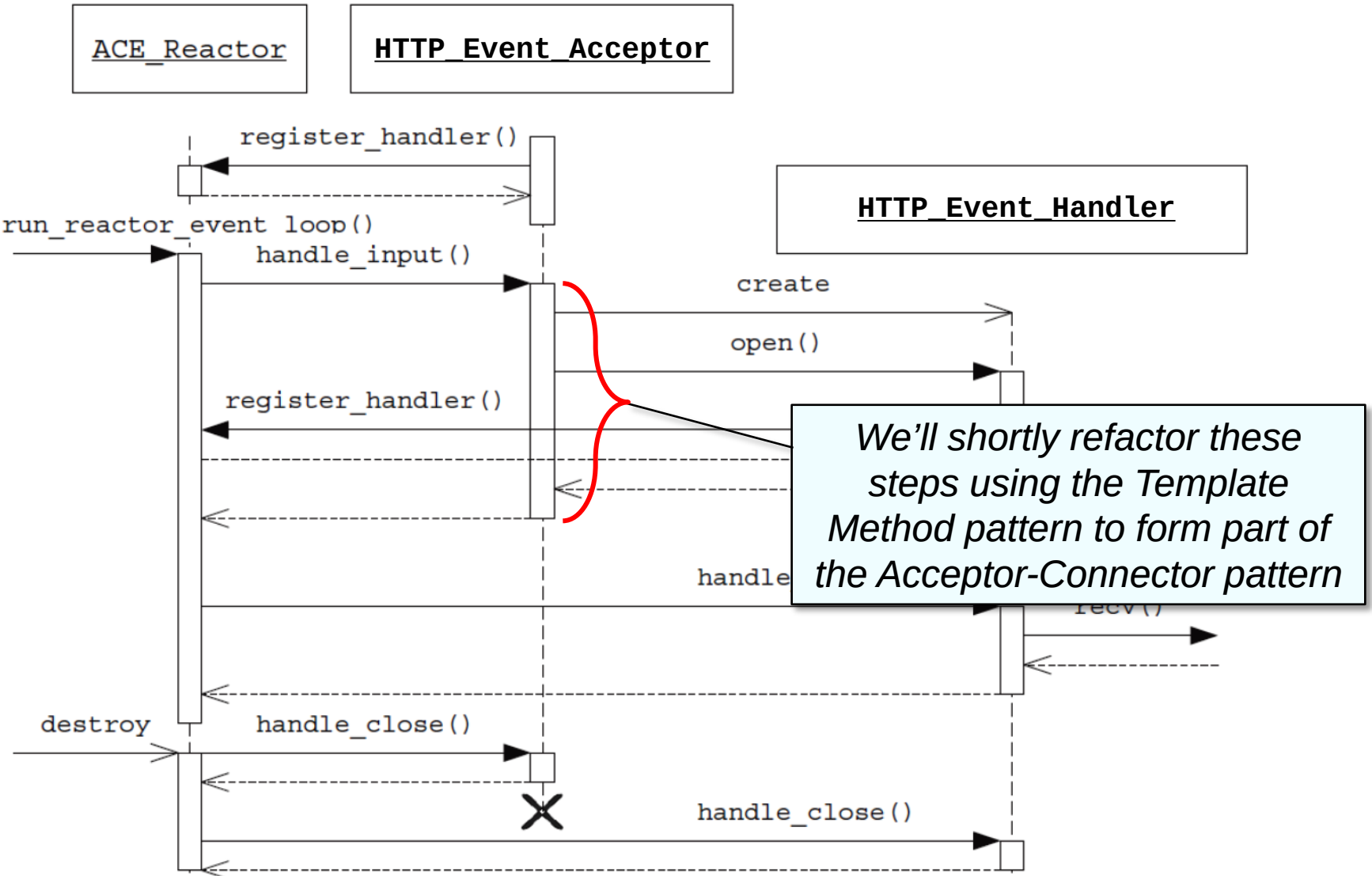
Sequence diagram for ACE *Reactor*-based web server

Using ACE_Reactor with JAWS



Sequence diagram for ACE *Reactor*-based web server

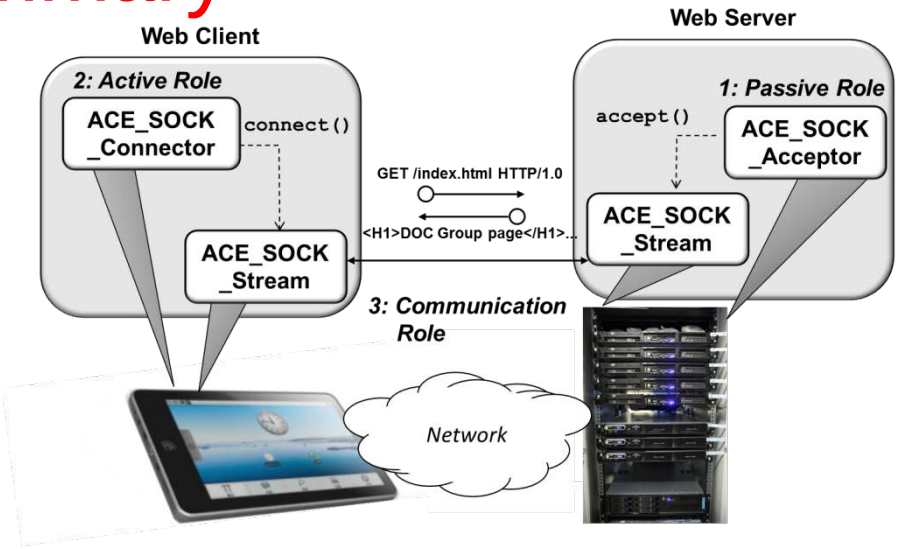
Using ACE_Reactor with JAWS



Sequence diagram for ACE *Reactor*-based web server

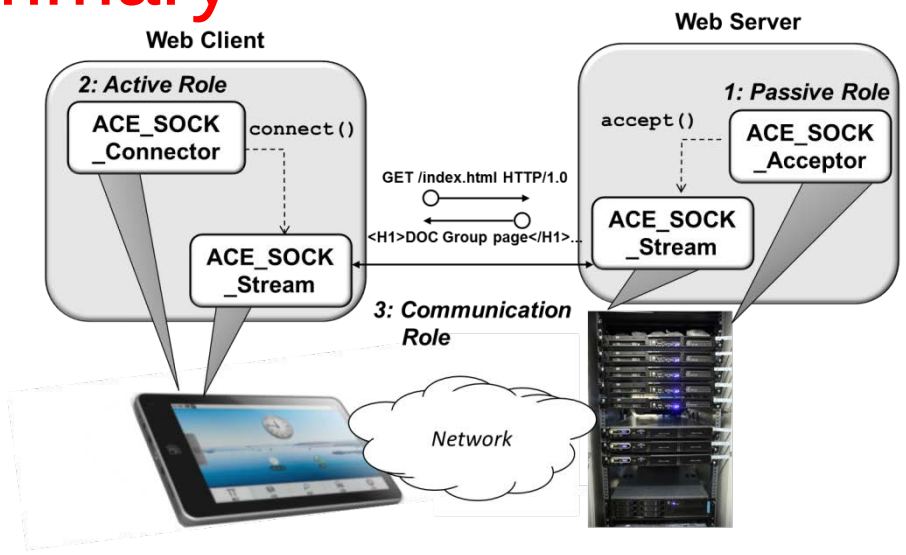
Summary

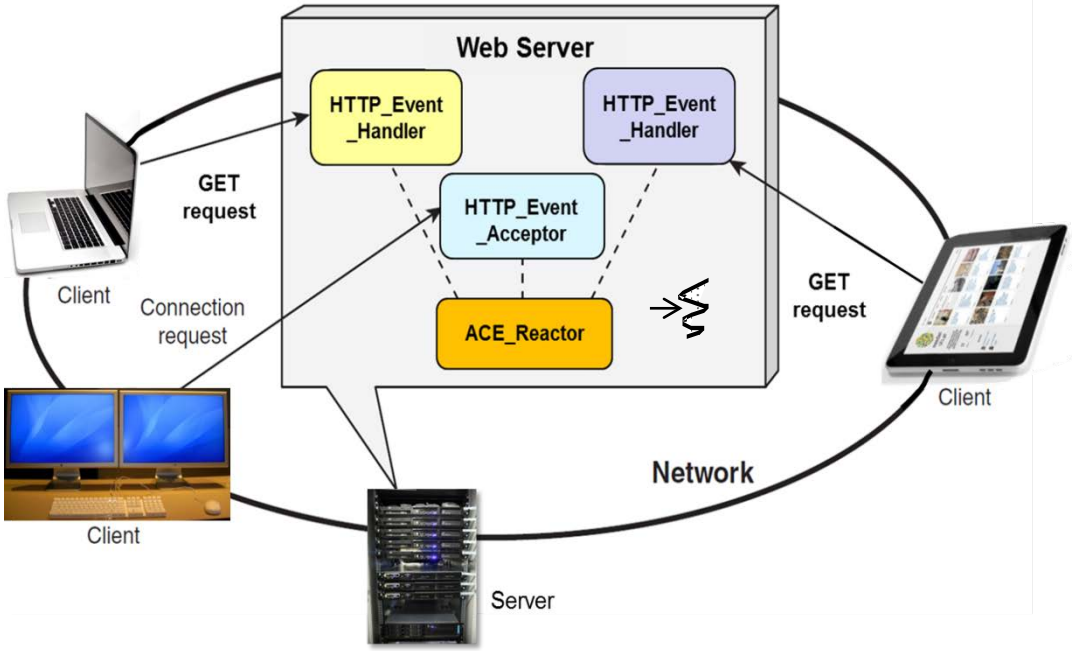
- The ACE *Reactor* version of JAWS is more modular & flexible than the *Wrapper Façade* version we described earlier
  - Wrapper facades resolve accidental complexities with system APIs, but don't address architectural concerns



Summary

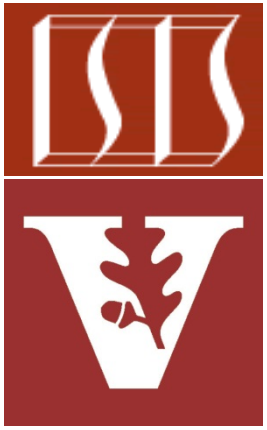
- The ACE *Reactor* version of JAWS is more modular & flexible than the *Wrapper Façade* version we described earlier
  - Wrapper facades resolve accidental complexities with system APIs, but don't address architectural concerns
- The Reactor framework provides the architecture for the JAWS reactive web server version
  - It's core processing logic is similar
  - The context in which this logic is based, however, can be adapted & extended much more readily





Patterns & Frameworks for Synchronous Event Handling, Connections, & Service Initialization: Part 4

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt



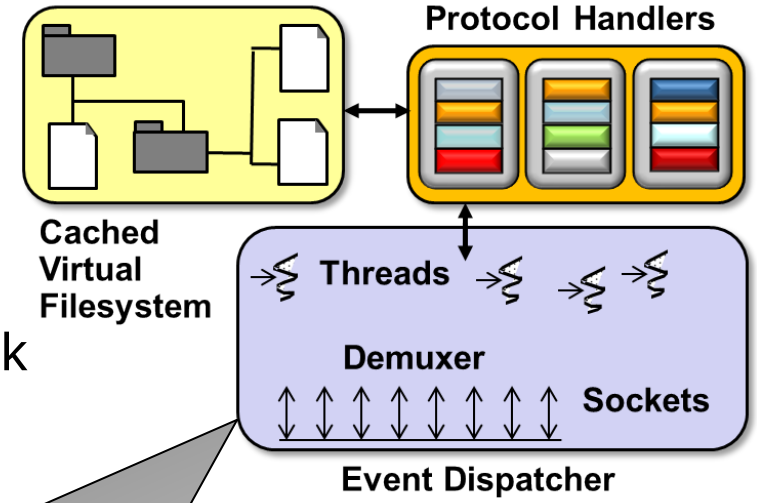
Professor of Computer Science

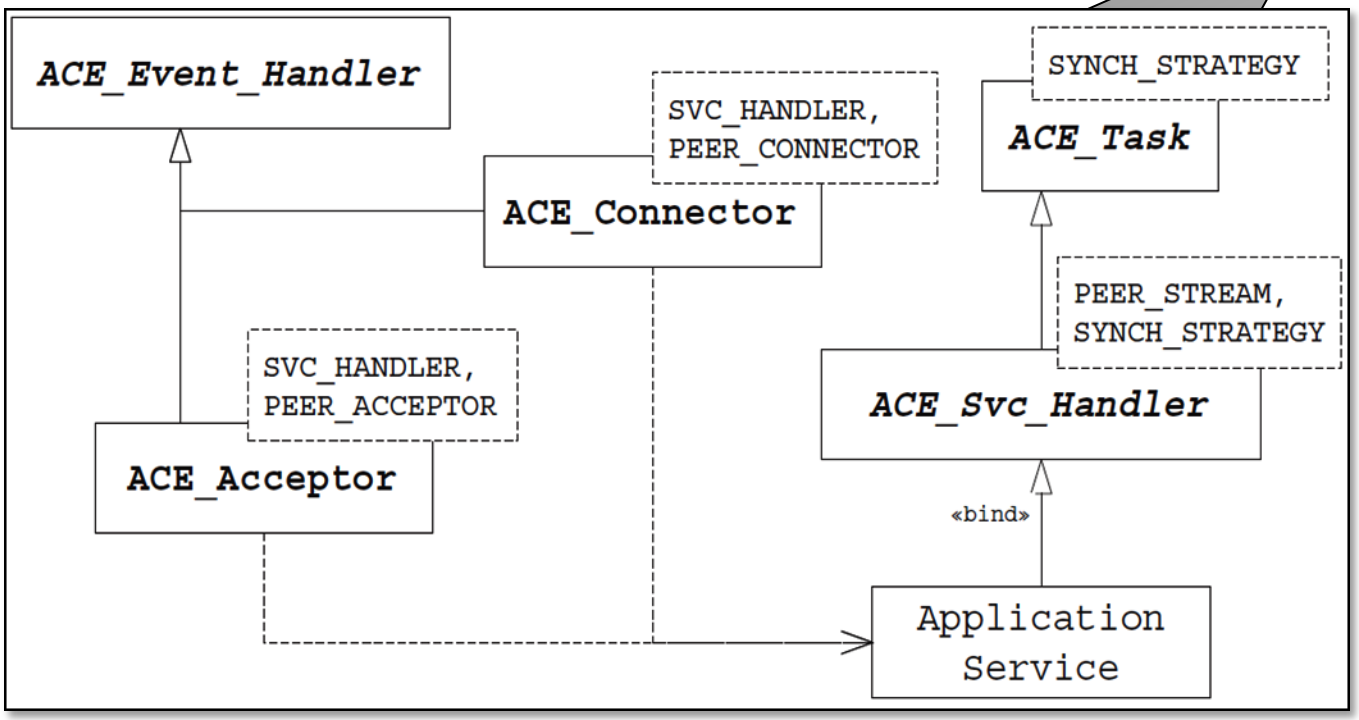
Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Topics Covered in this Part of the Module

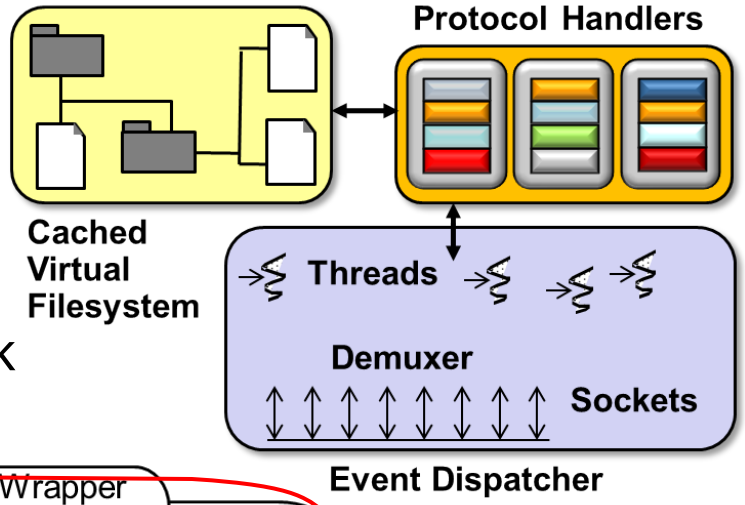
- Describe the *Reactor* & *Acceptor-Connector* patterns
- Describe the ACE *Reactor* framework
- Apply ACE *Reactor* to JAWS
- Describe ACE *Acceptor-Connector* framework & apply it to JAWS

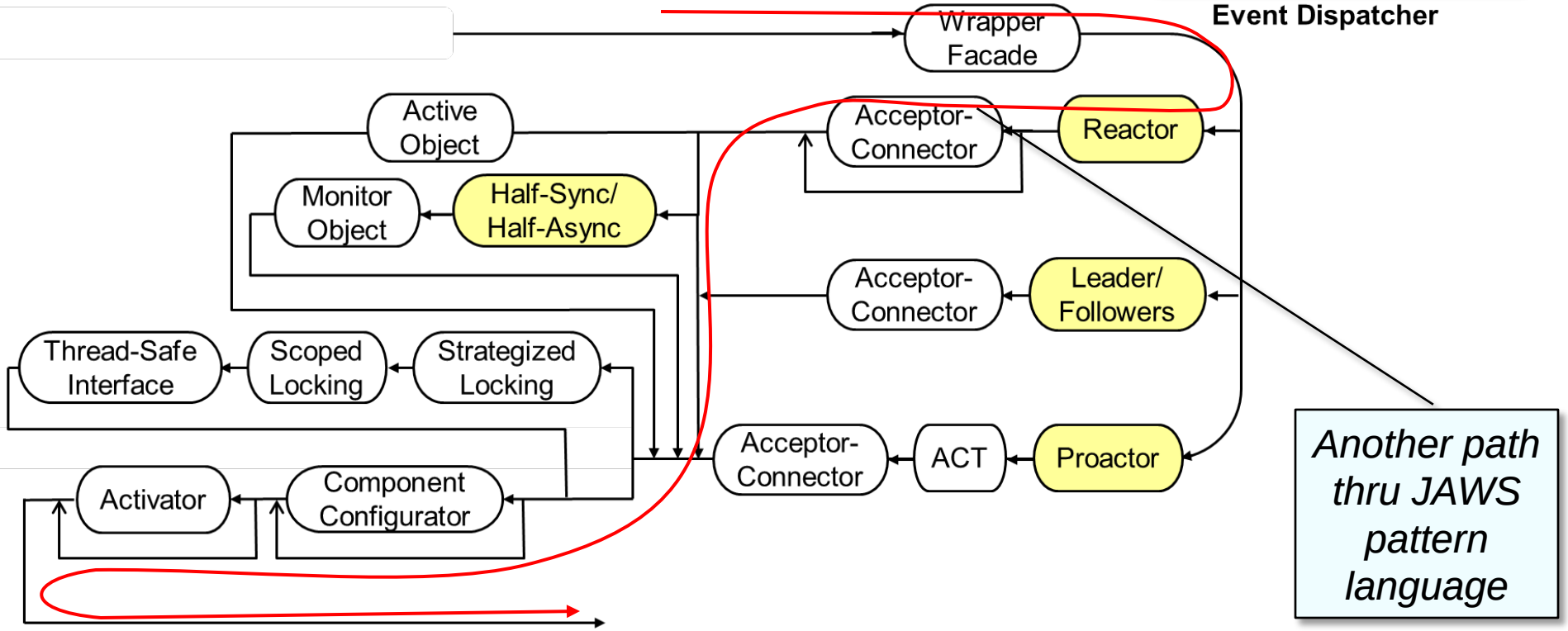




Topics Covered in this Part of the Module

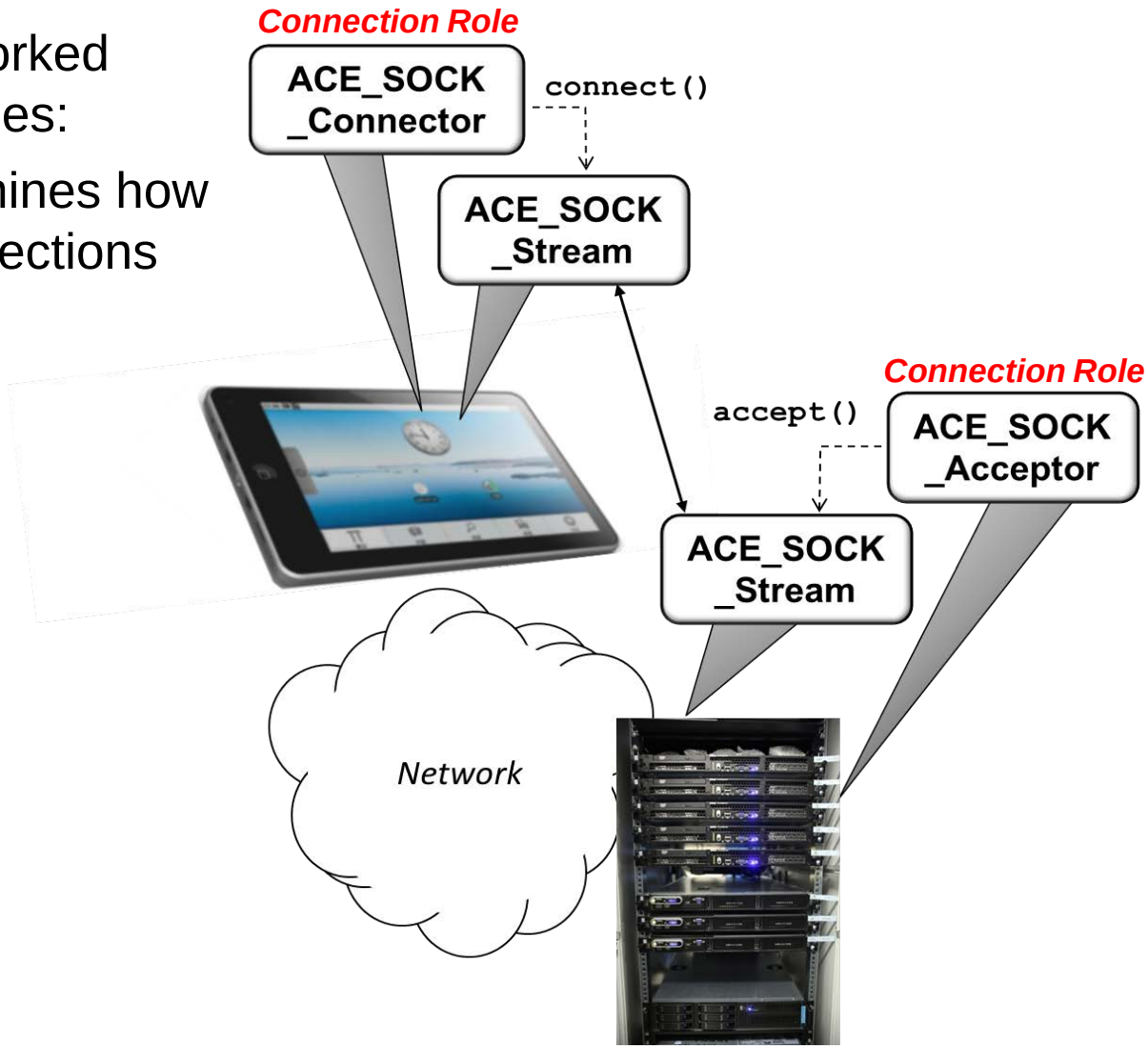
- Describe the *Reactor* & *Acceptor-Connector* patterns
- Describe the ACE *Reactor* framework
- Apply ACE *Reactor* to JAWS
- Describe ACE *Acceptor-Connector* framework & apply it to JAWS





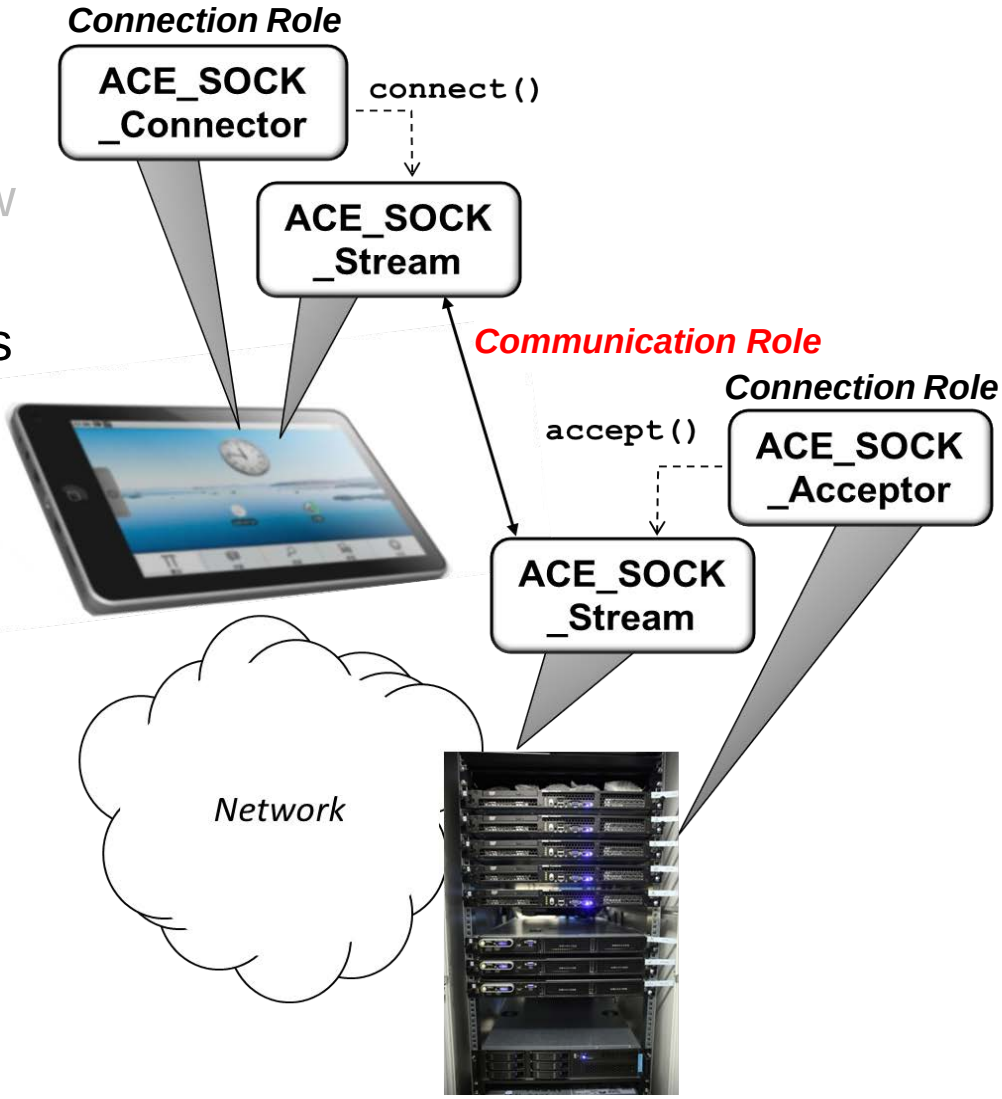
Motivation for ACE Acceptor-Connector Framework

- Connection-oriented networked apps often play stylized roles:
  - **Connection role** determines how an app establishes connections



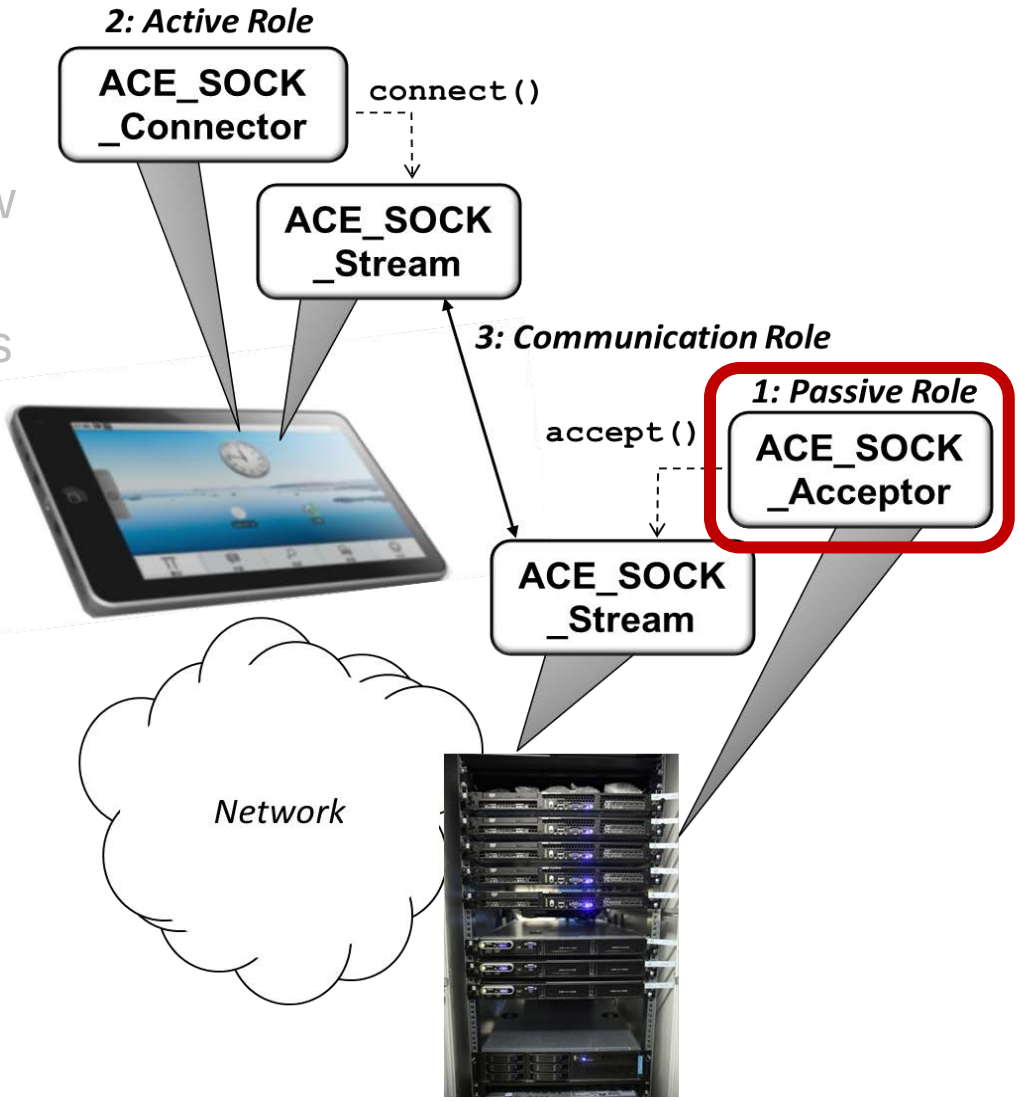
Motivation for ACE Acceptor-Connector Framework

- Connection-oriented networked apps often play stylized roles:
  - **Connection role** determines how an app establishes connections
  - **Communication role** determines whether an app plays the role of a client, a server, or both



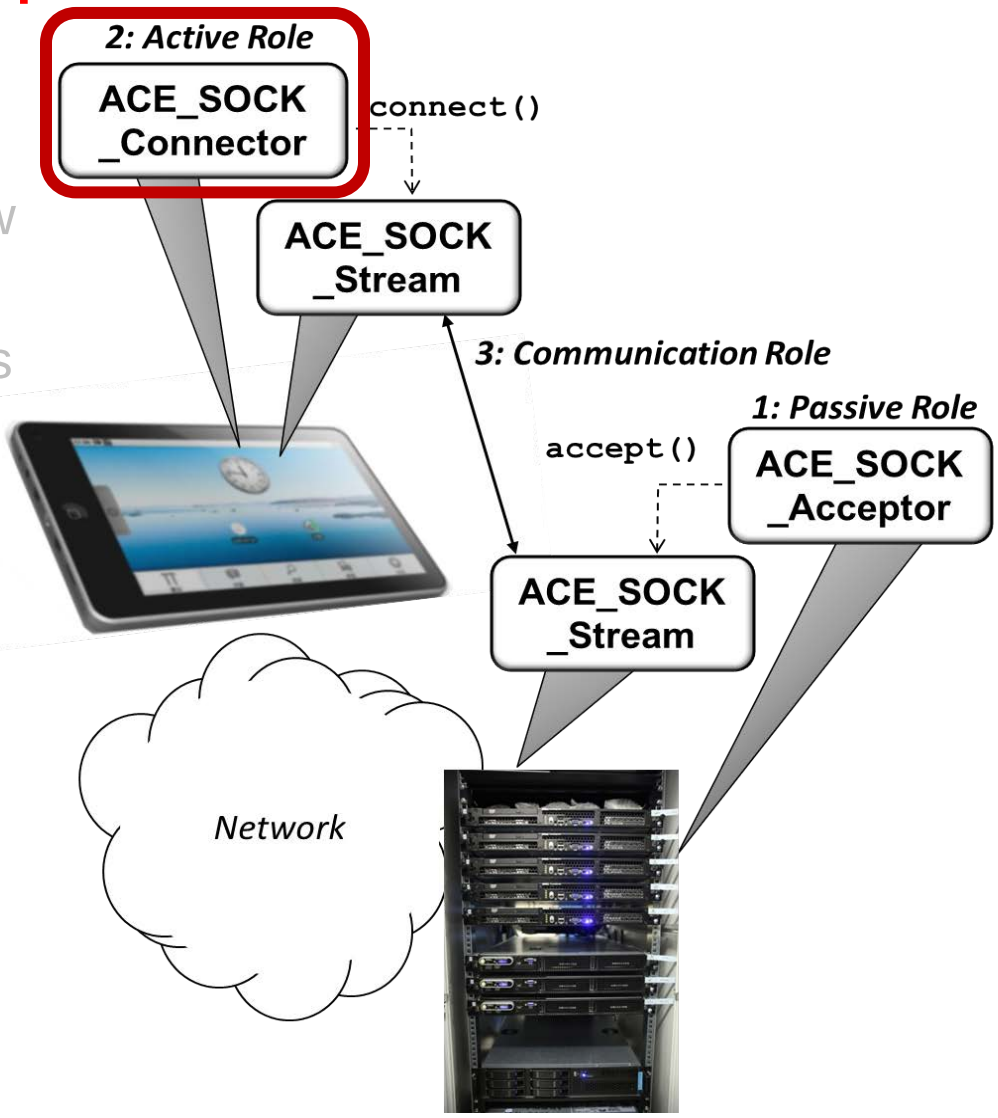
Motivation for ACE Acceptor-Connector Framework

- Connection-oriented networked apps often play stylized roles:
  - **Connection role** determines how an app establishes connections
  - **Communication role** determines whether an app plays the role of a client, a server, or both
- Connection-oriented networked apps typified by asymmetric connection roles:
  - Servers often *passively* accept connections at a listening port



Motivation for ACE Acceptor-Connector Framework

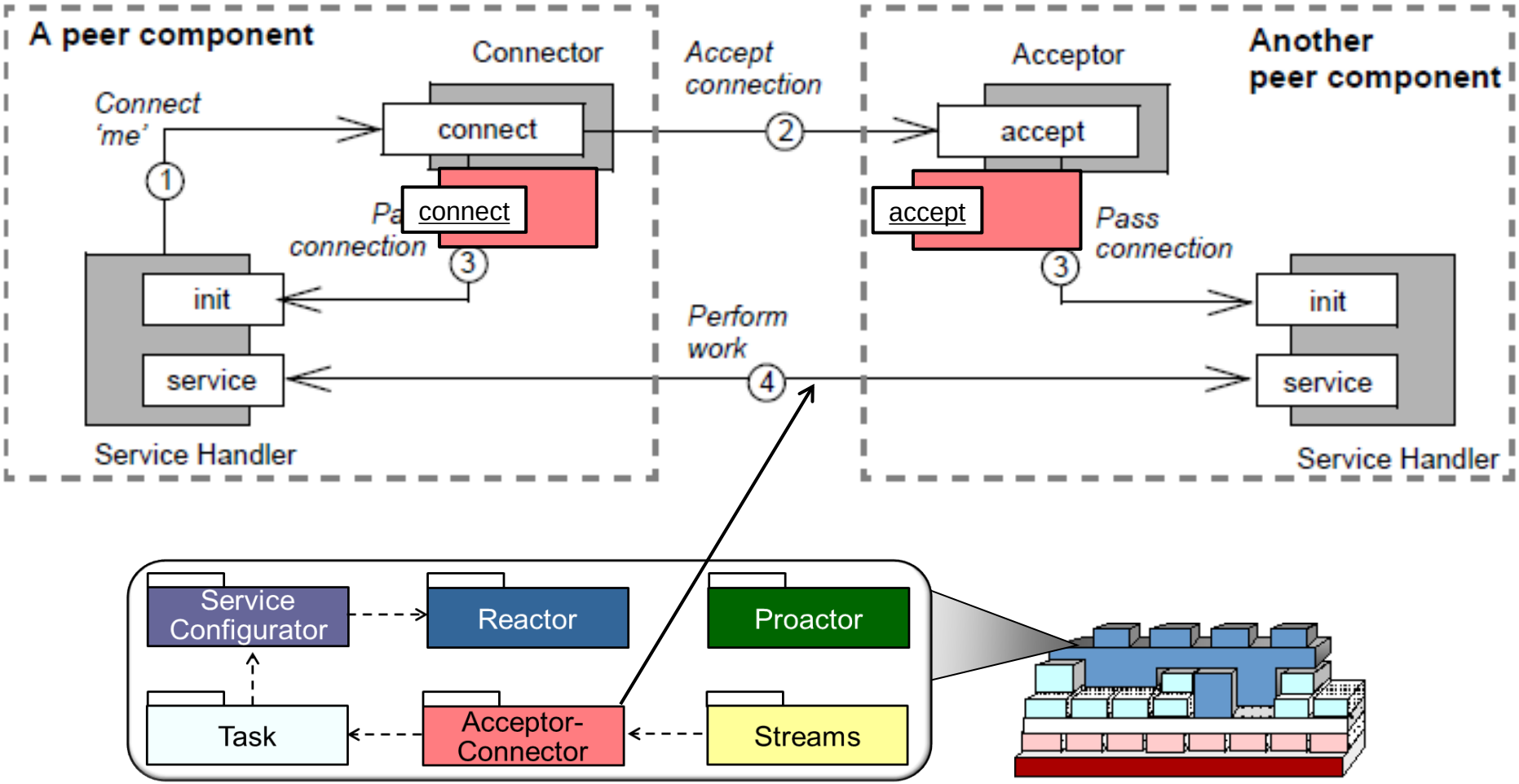
- Connection-oriented networked apps often play stylized roles:
  - **Connection role** determines how an app establishes connections
  - **Communication role** determines whether an app plays the role of a client, a server, or both
- Connection-oriented networked apps typified by asymmetric connection roles:
  - Servers often *passively* accept connections at a listening port
  - Clients often *actively* initiate connections by connecting to a server's listening port



Factoring out these roles into a framework can enhance reuse & robustness

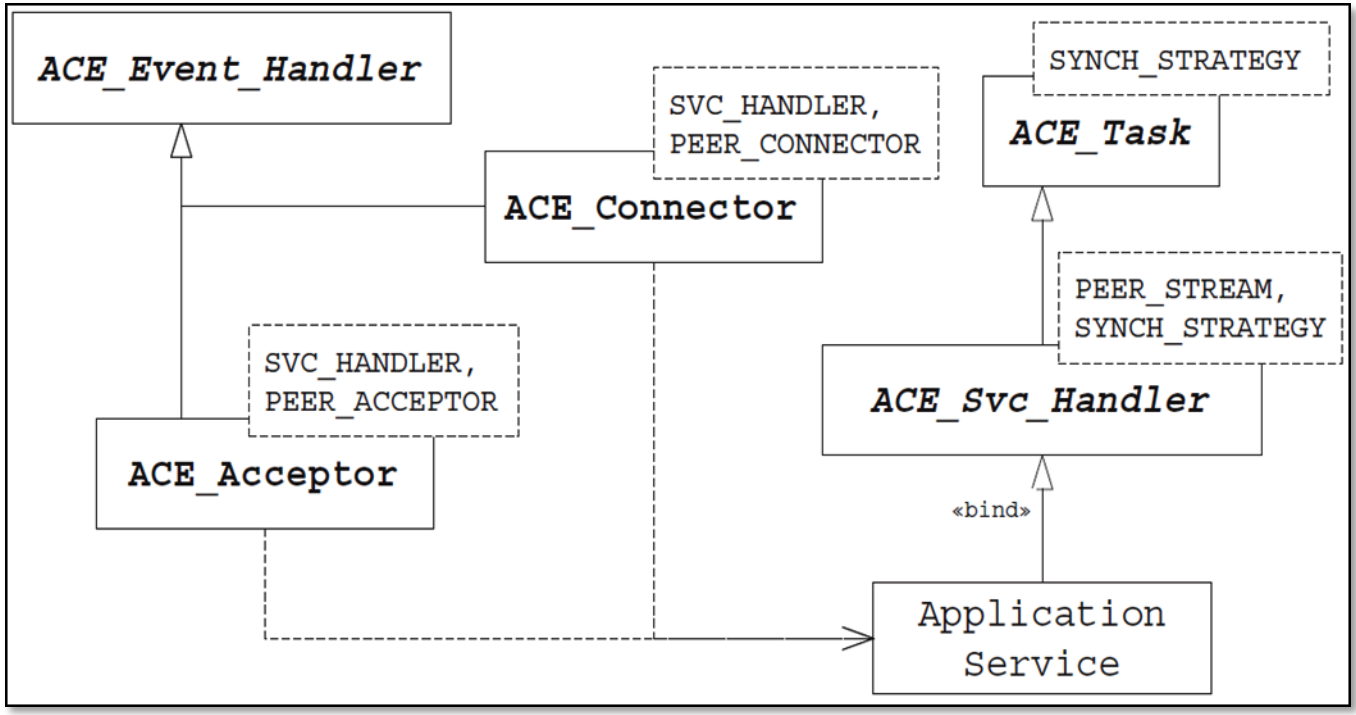
Overview of ACE Acceptor-Connector Framework

- The classes in this framework decouple connection & initialization of cooperating peer services in a networked app from processing they perform after being connected & initialized



Overview of ACE Acceptor-Connector Framework

- The classes in this framework decouple connection & initialization of cooperating peer services in a networked app from processing they perform after being connected & initialized
- Apps subclass from **`ACE_Svc_Handler`** & override its hook methods to process events via various activation models
  - e.g., reactive & multi-threading/ multi-processing



Classes are designed in accordance with the *Acceptor-Connector* pattern

Overview of ACE Acceptor-Connector Framework

- The classes in this framework decouple connection & initialization of cooperating peer services in a networked app from processing they perform after being connected & initialized
- Apps subclass from **`ACE_Svc_Handler`** & override its hook methods to process events via various activation models
- Key classes in the ACE *Acceptor-Connector* framework include:

ACE Class	Description
`ACE_Svc_Handler`	Represents the local end of a connected service & contains an IPC endpoint used to communicate with a connected peer
`ACE_Acceptor`	This factory waits passively to accept a connection and then initializes an **`ACE Svc Handler`** in response to an active connection request from a peer
`ACE_Connector`	This factory actively connects to a peer acceptor and then initializes an **`ACE_Svc_Handler`** to communicate with its connected peer

Overview of ACE Acceptor-Connector Framework

- The classes in this framework decouple connection & initialization of cooperating peer services in a networked app from processing they perform after being connected & initialized
- Apps subclass from **`ACE_Svc_Handler`** & override its hook methods to process events via various activation models
- Key classes in the ACE *Acceptor-Connector* framework include:

ACE Class	Description
`ACE_Svc_Handler`	Represents the local end of a connected service & contains an IPC endpoint used to communicate with a connected peer
`ACE_Acceptor`	This factory waits passively to accept a connection and then initializes an **`ACE_Svc_Handler`** in response to an active connection request from a peer
`ACE_Connector`	This factory actively connects to a peer acceptor and then initializes an **`ACE_Svc_Handler`** to communicate with its connected peer

Overview of ACE Acceptor-Connector Framework

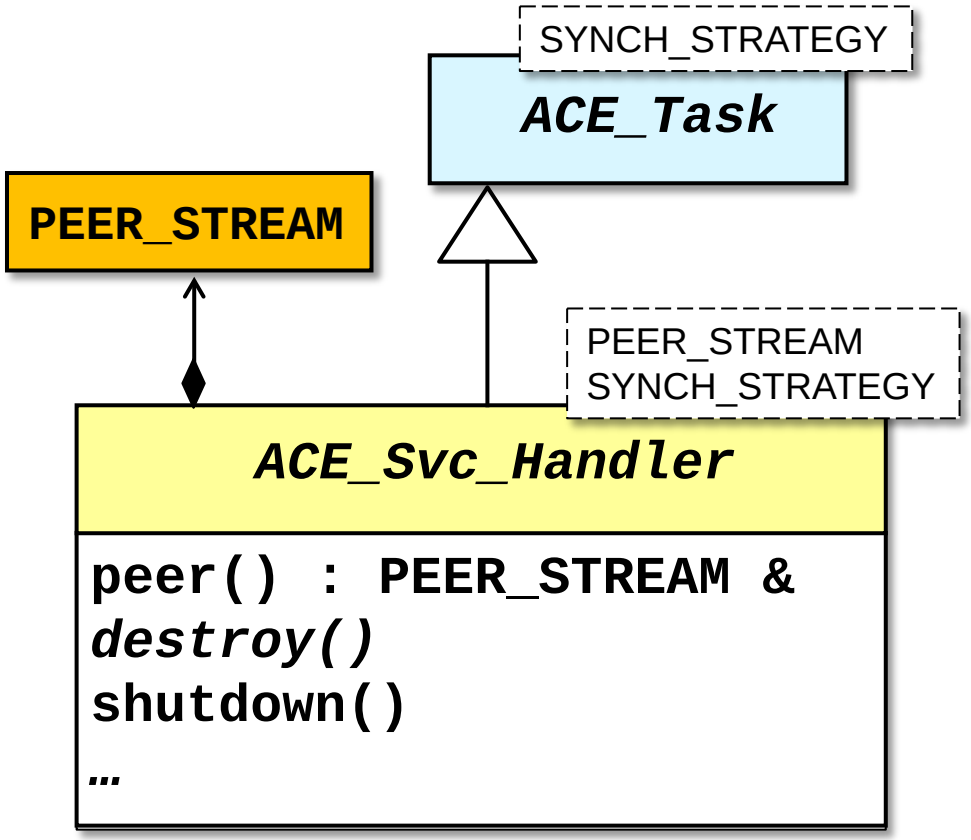
- The classes in this framework decouple connection & initialization of cooperating peer services in a networked app from processing they perform after being connected & initialized
- Apps subclass from **`ACE_Svc_Handler`** & override its hook methods to process events via various activation models
- Key classes in the ACE *Acceptor-Connector* framework include:

ACE Class	Description
`ACE_Svc_Handler`	Represents the local end of a connected service & contains an IPC endpoint used to communicate with a connected peer
`ACE_Acceptor`	This factory waits passively to accept a connection and then initializes an **`ACE_Svc_Handler`** in response to an active connection request from a peer
`ACE_Connector`	This factory actively connects to a peer acceptor and then initializes an **`ACE_Svc_Handler`** to communicate with its connected peer

See www.dre.vanderbilt.edu/~schmidt/PDF/Acc-Con.pdf for more info

The ACE_Svc_Handler Class

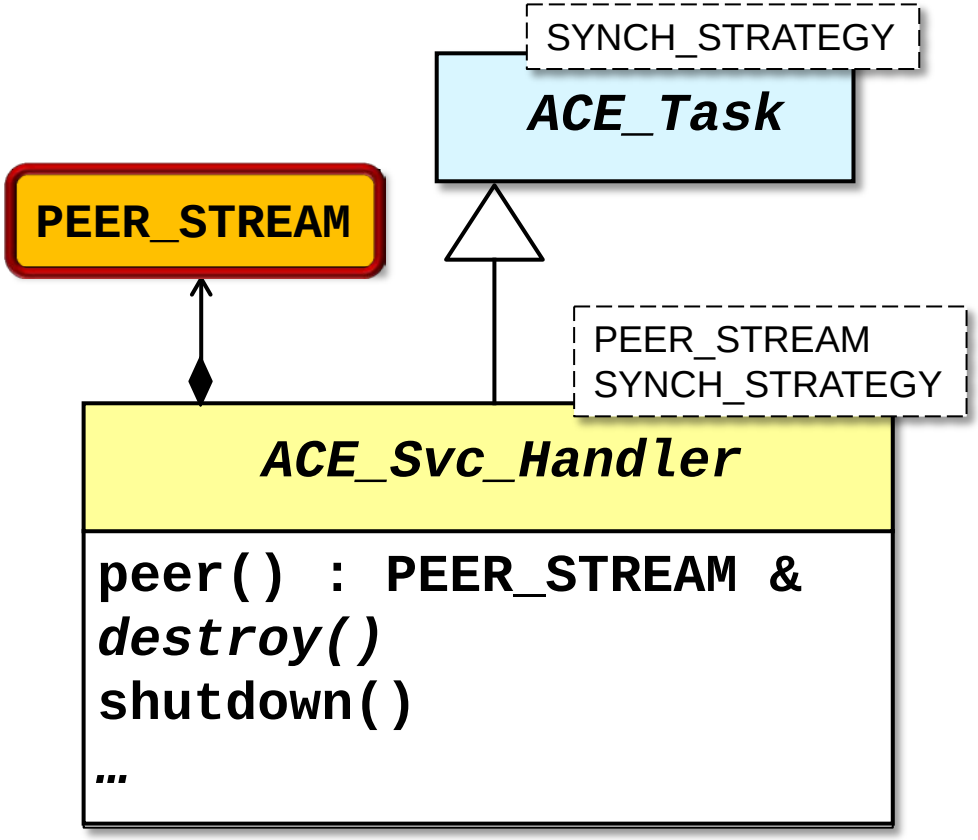
This class provides synchronous & reactive data transfer capabilities:



The ACE_Svc_Handler Class

This class provides synchronous & reactive data transfer capabilities:

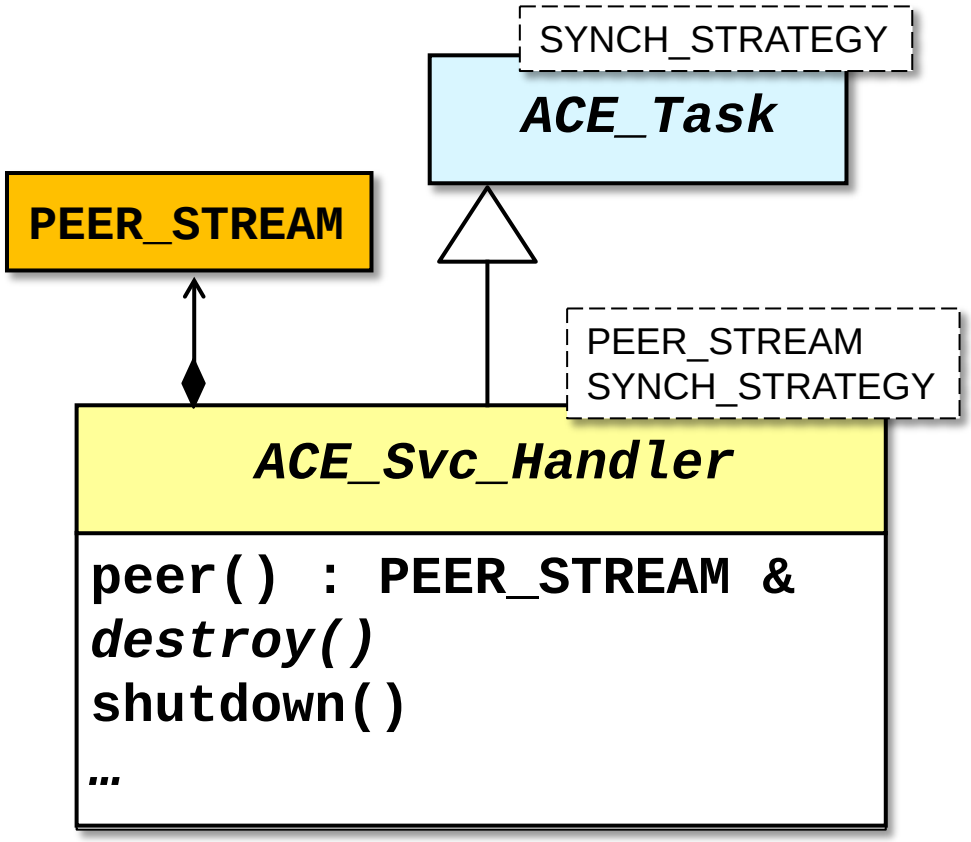
- It provides an IPC endpoint used to communicate with its peer service handler



The ACE_Svc_Handler Class

This class provides synchronous & reactive data transfer capabilities:

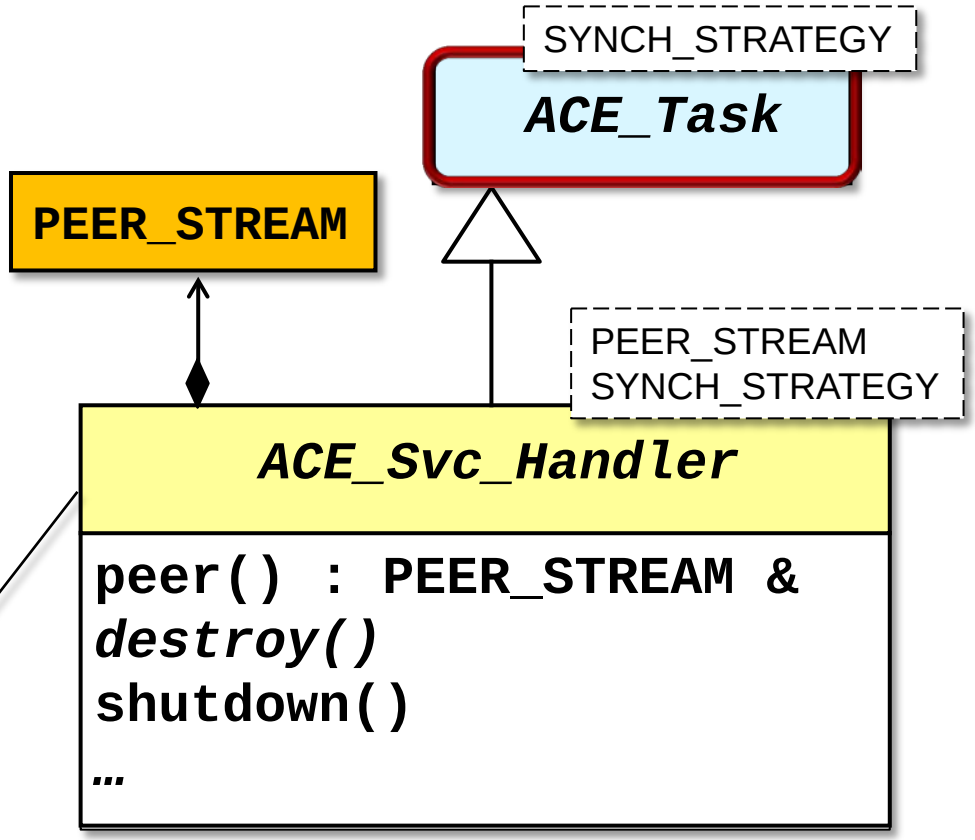
- It provides an IPC endpoint used to communicate with its peer service handler
- It codifies common practices of reactive network services
  - e.g., registering with a reactor when service is opened & closing IPC endpoint when unregistering a service from a reactor



The ACE_Svc_Handler Class

This class provides synchronous & reactive data transfer capabilities:

- It provides an IPC endpoint used to communicate with its peer service handler
- It codifies common practices of reactive network services
  - e.g., registering with a reactor when service is opened & closing IPC endpoint when unregistering a service from a reactor
- It can also participate in other concurrency models
  - e.g., active object, etc.

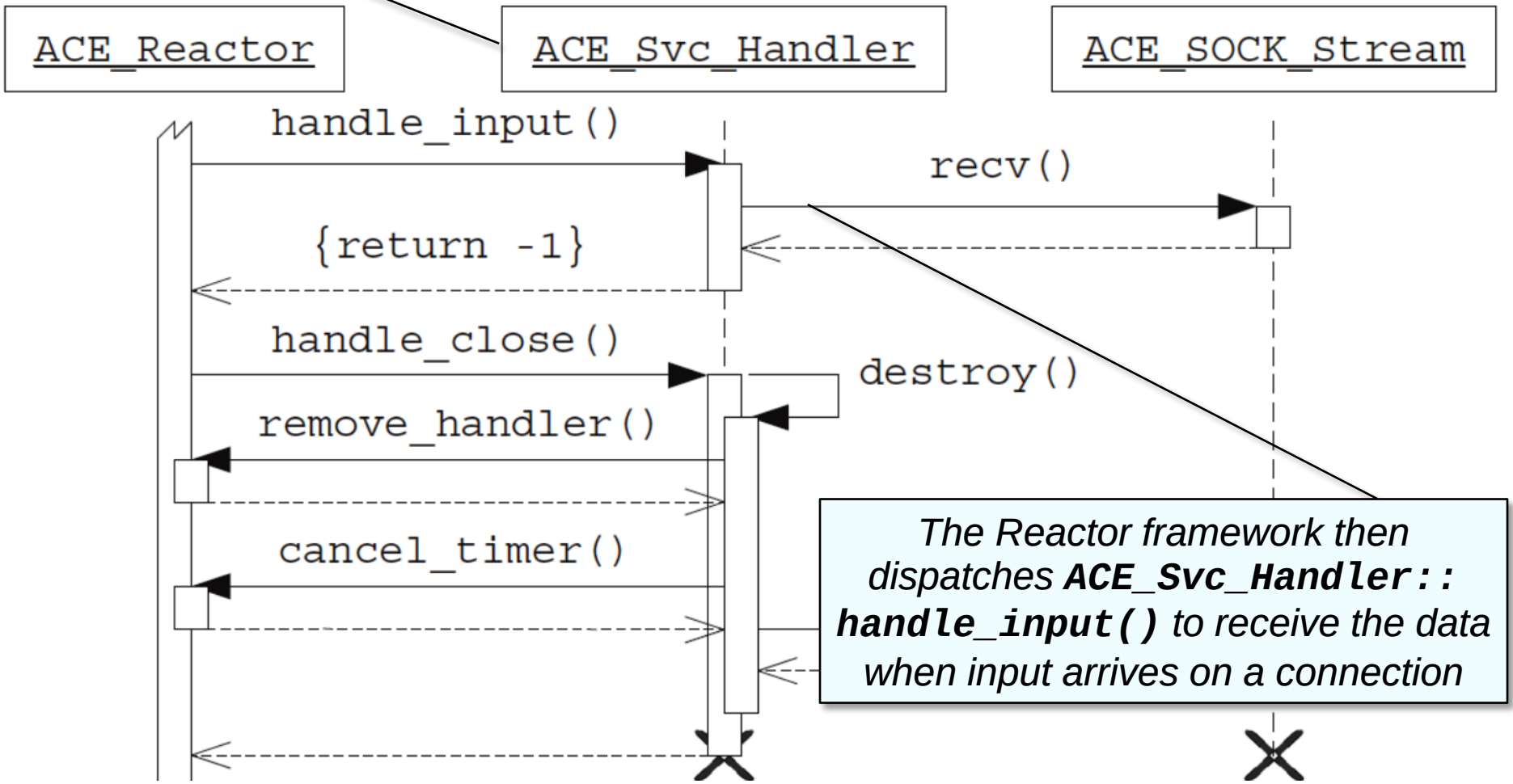


ACE_Svc_Handler *inherits ACE concurrency, queueing, synchronization, dynamic configuration, & event handling framework capabilities from its base classes*

Handles *variability* of IPC & synchronization via a *common* network I/O API

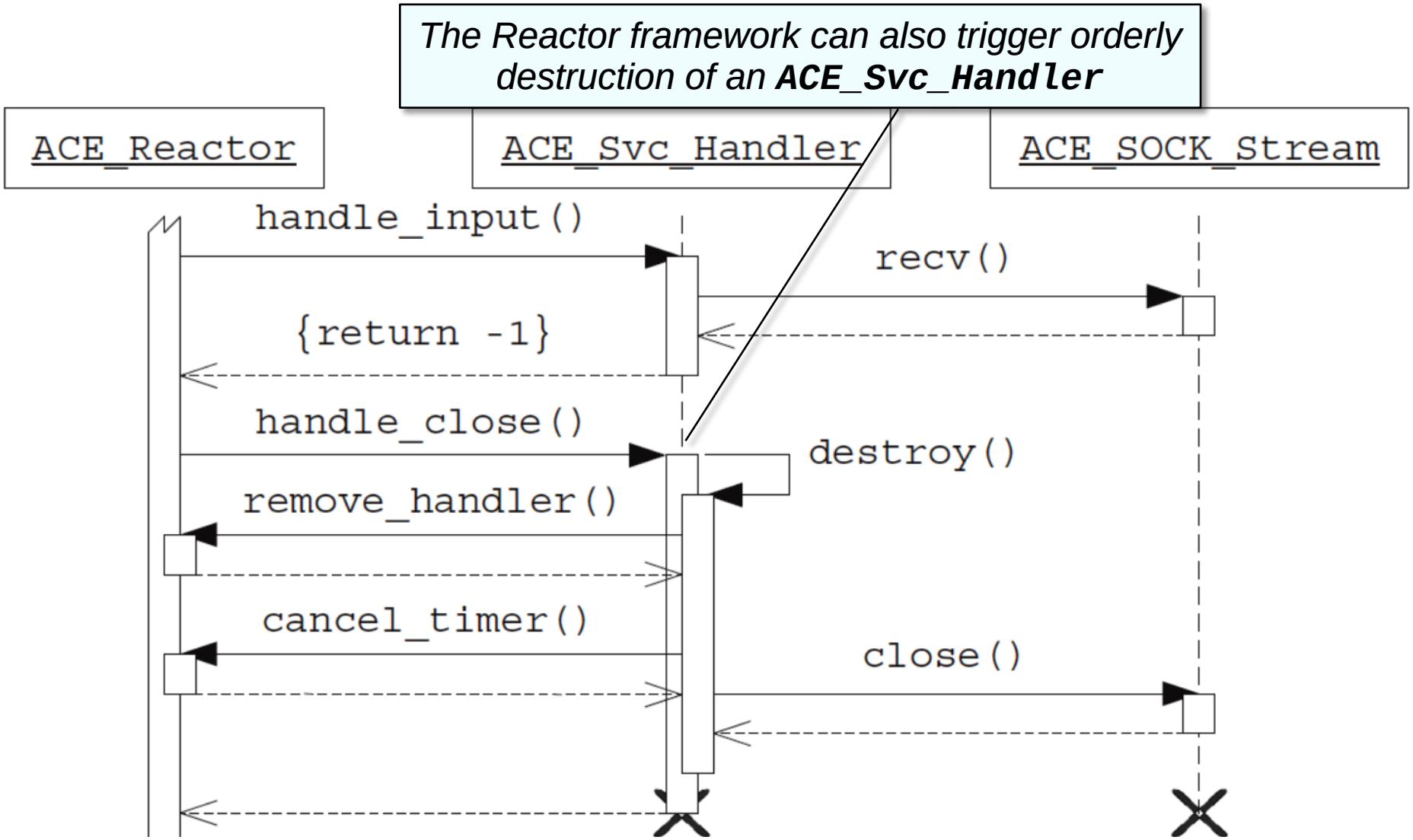
Combining ACE_Svc_Handler with ACE_Reactor

*An **`ACE_Svc_Handler`** can be registered with the ACE Reactor framework for READ events*



*The Reactor framework then dispatches **`ACE_Svc_Handler::handle_input()`** to receive the data when input arrives on a connection*

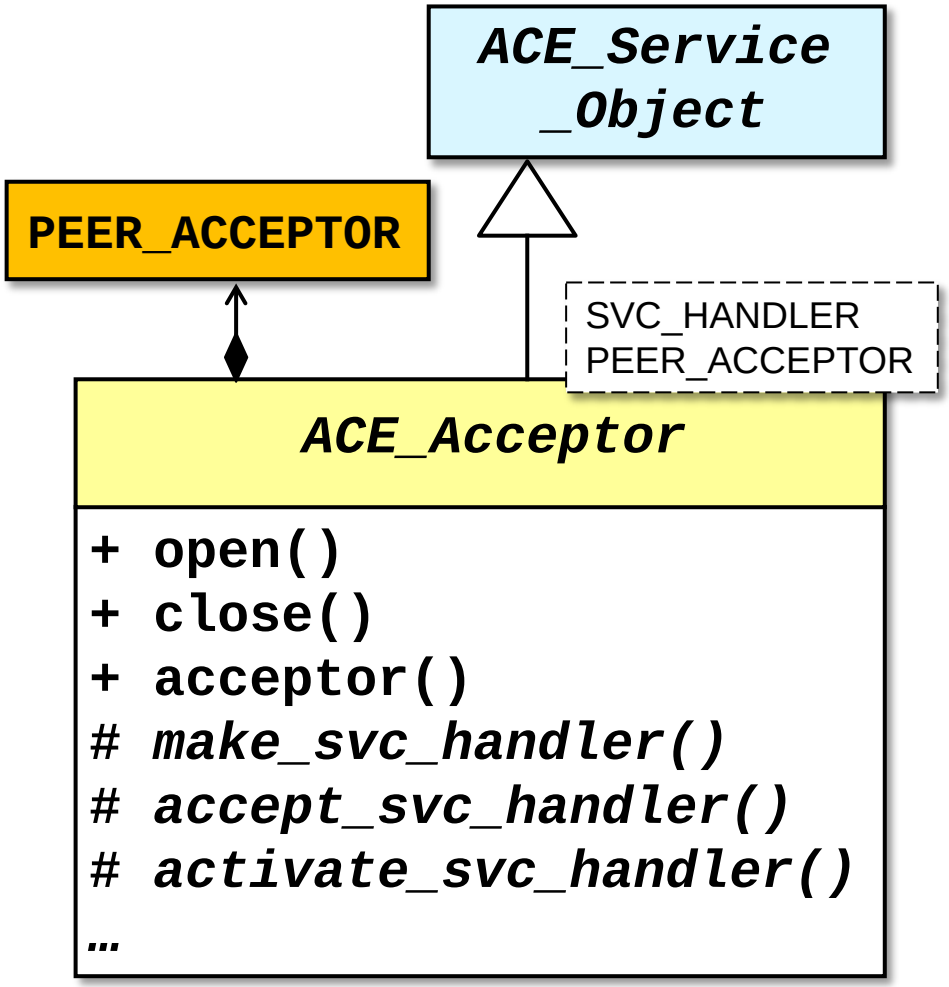
Combining ACE_Svc_Handler with ACE_Reactor



The ACE_Acceptor Class

This class is a factory playing the *Acceptor* role in *Acceptor-Connector* pattern:

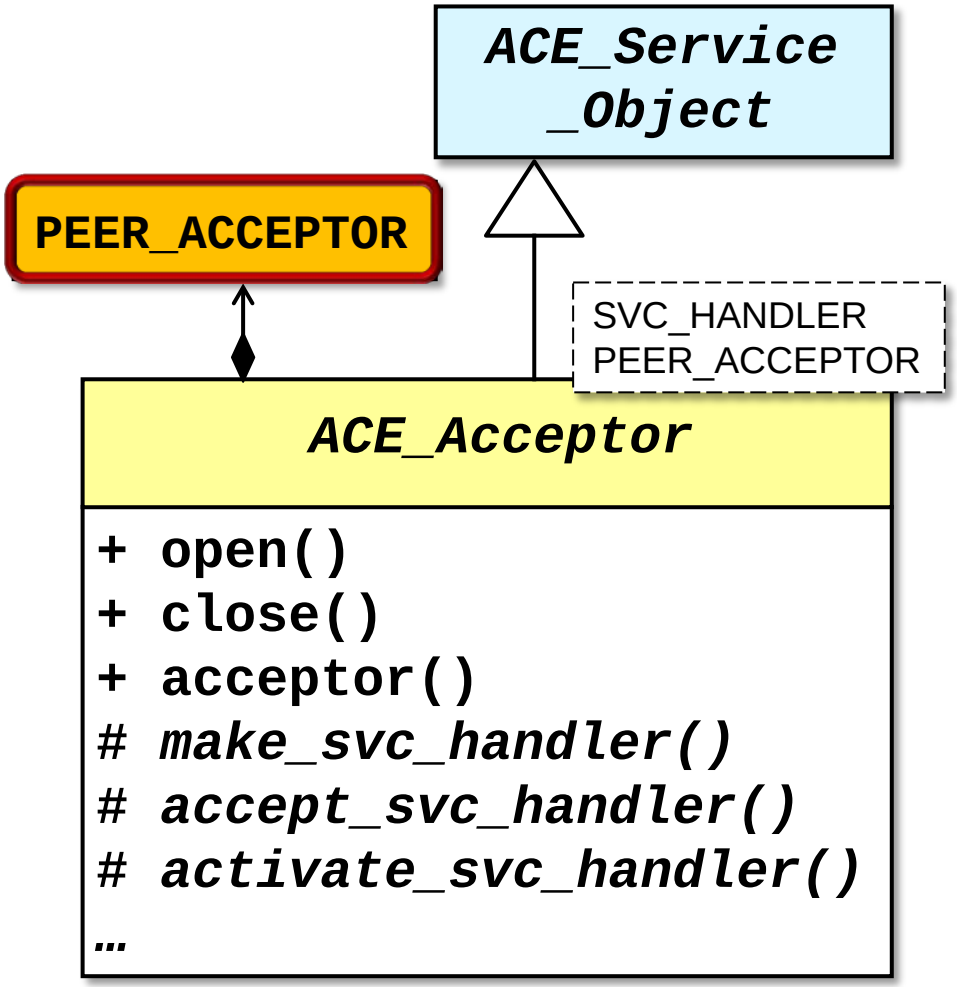
- It decouples passive connection establishment & service initialization logic from subsequent processing performed by a service handler



The ACE_Acceptor Class

This class is a factory playing the *Acceptor* role in *Acceptor-Connector* pattern:

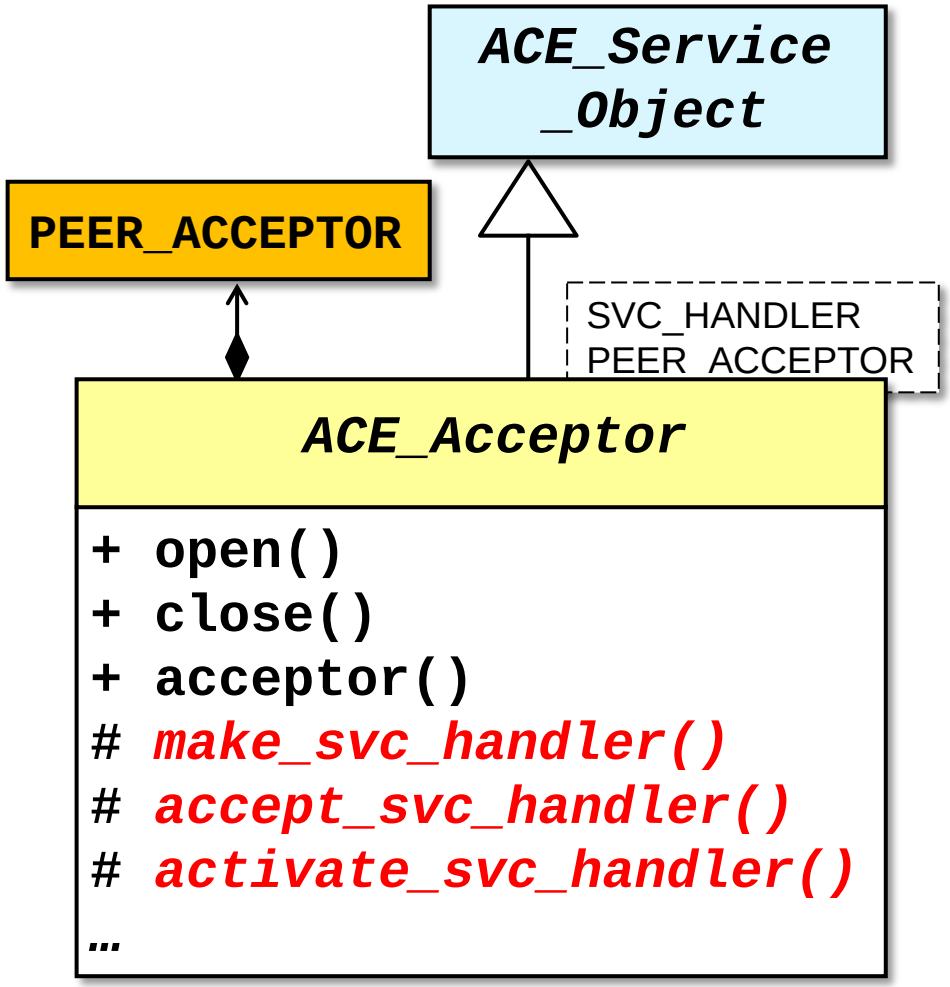
- It decouples passive connection establishment & service initialization logic from subsequent processing performed by a service handler
- It provides a parameterized passive-mode IPC endpoint that listens for & accepts connections from peers



The ACE_Acceptor Class

This class is a factory playing the *Acceptor* role in *Acceptor-Connector* pattern:

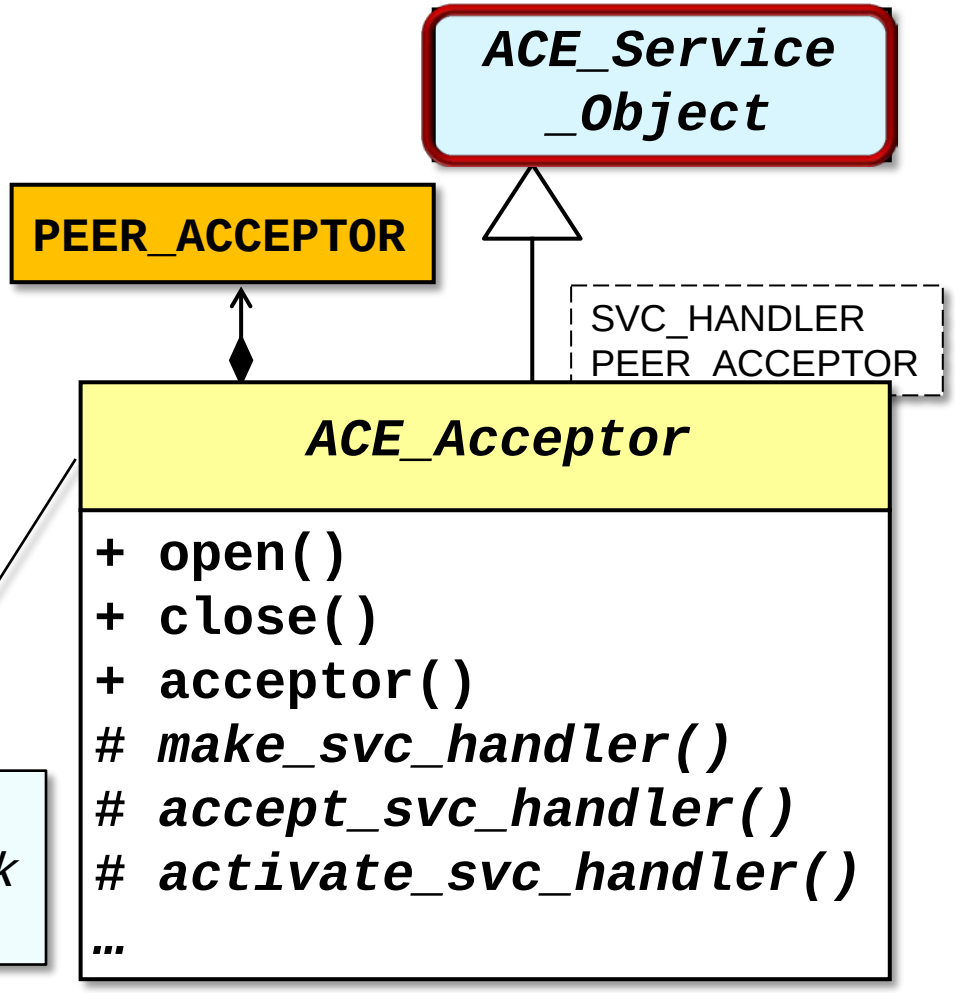
- It decouples passive connection establishment & service initialization logic from subsequent processing performed by a service handler
- It provides a parameterized passive-mode IPC endpoint that listens for & accepts connections from peers
- It automates the steps necessary to connect the IPC endpoint passively & create/activate its associated service handler



The ACE_Acceptor Class

This class is a factory playing the *Acceptor* role in *Acceptor-Connector* pattern:

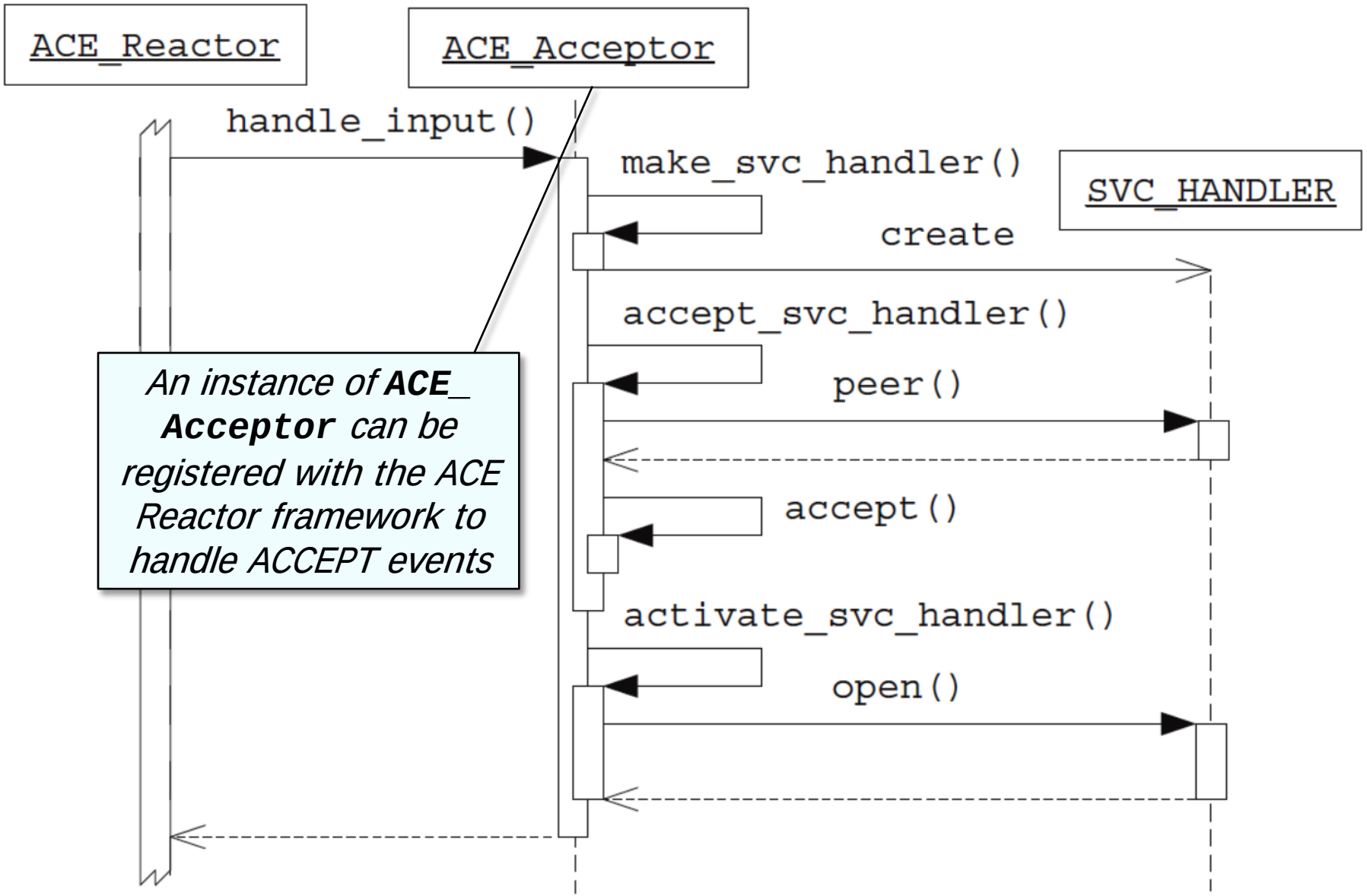
- It decouples passive connection establishment & service initialization logic from subsequent processing performed by a service handler
- It provides a parameterized passive-mode IPC endpoint that listens for & accepts connections from peers
- It automates the steps necessary to connect the IPC endpoint passively & create/activate its associated service handler



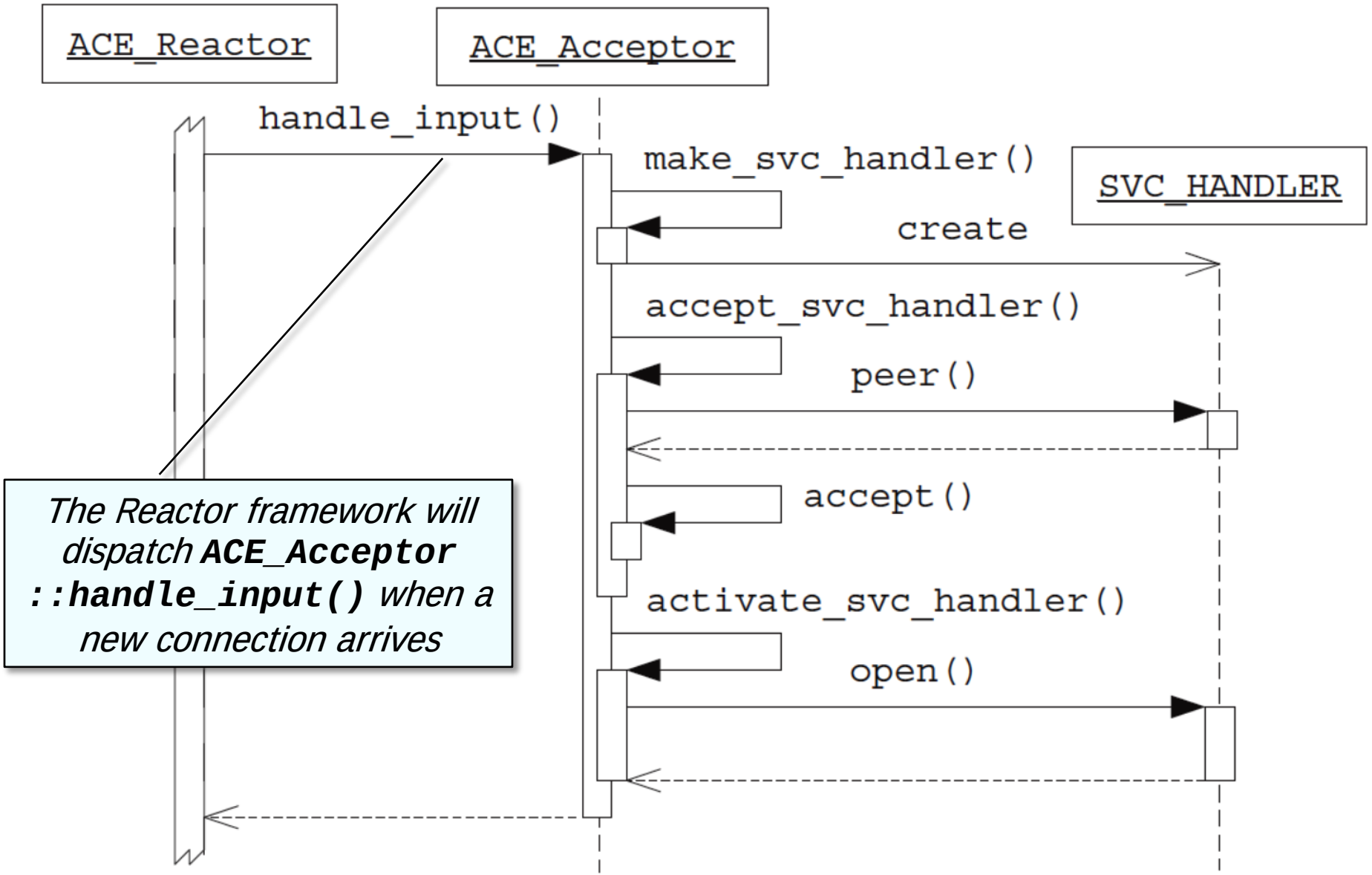
ACE_Acceptor *inherits ACE dynamic configuration & event handling framework capabilities from its base classes*

Handles IPC & service handler *variability* via *common* connect/initialize API

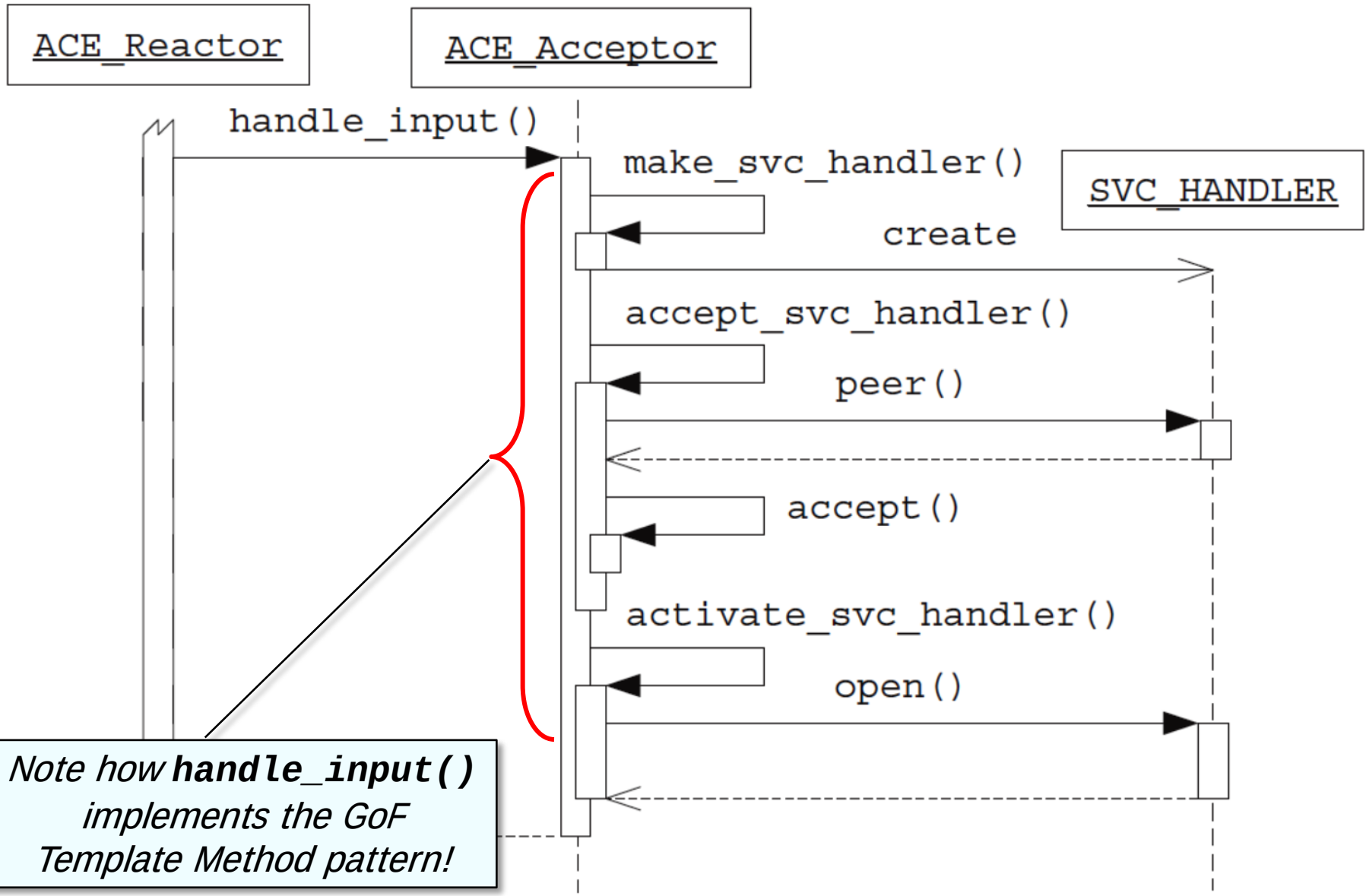
Combining ACE_Acceptor with ACE_Reactor



Combining ACE_Acceptor with ACE_Reactor

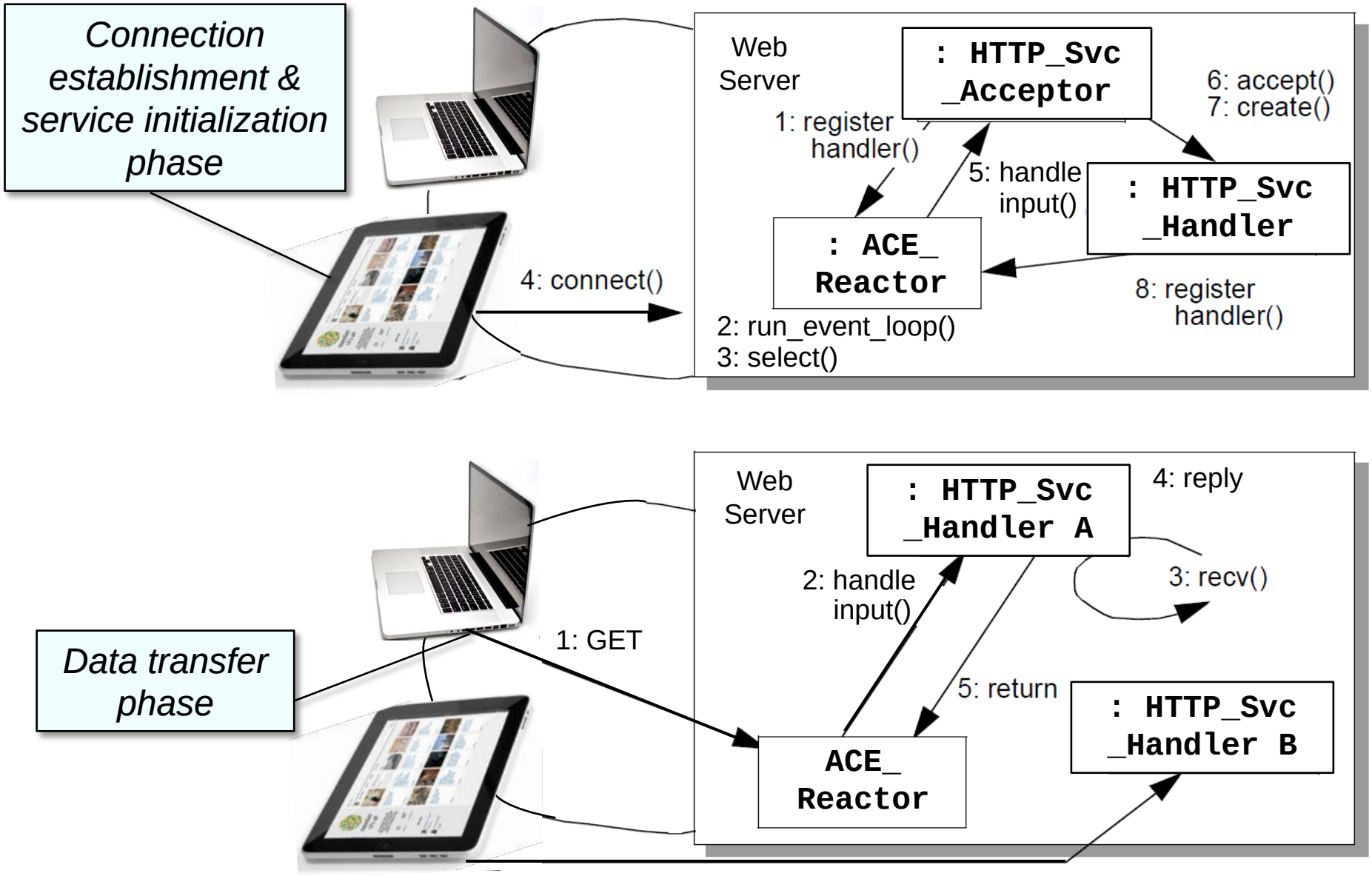


Combining ACE_Acceptor with ACE_Reactor



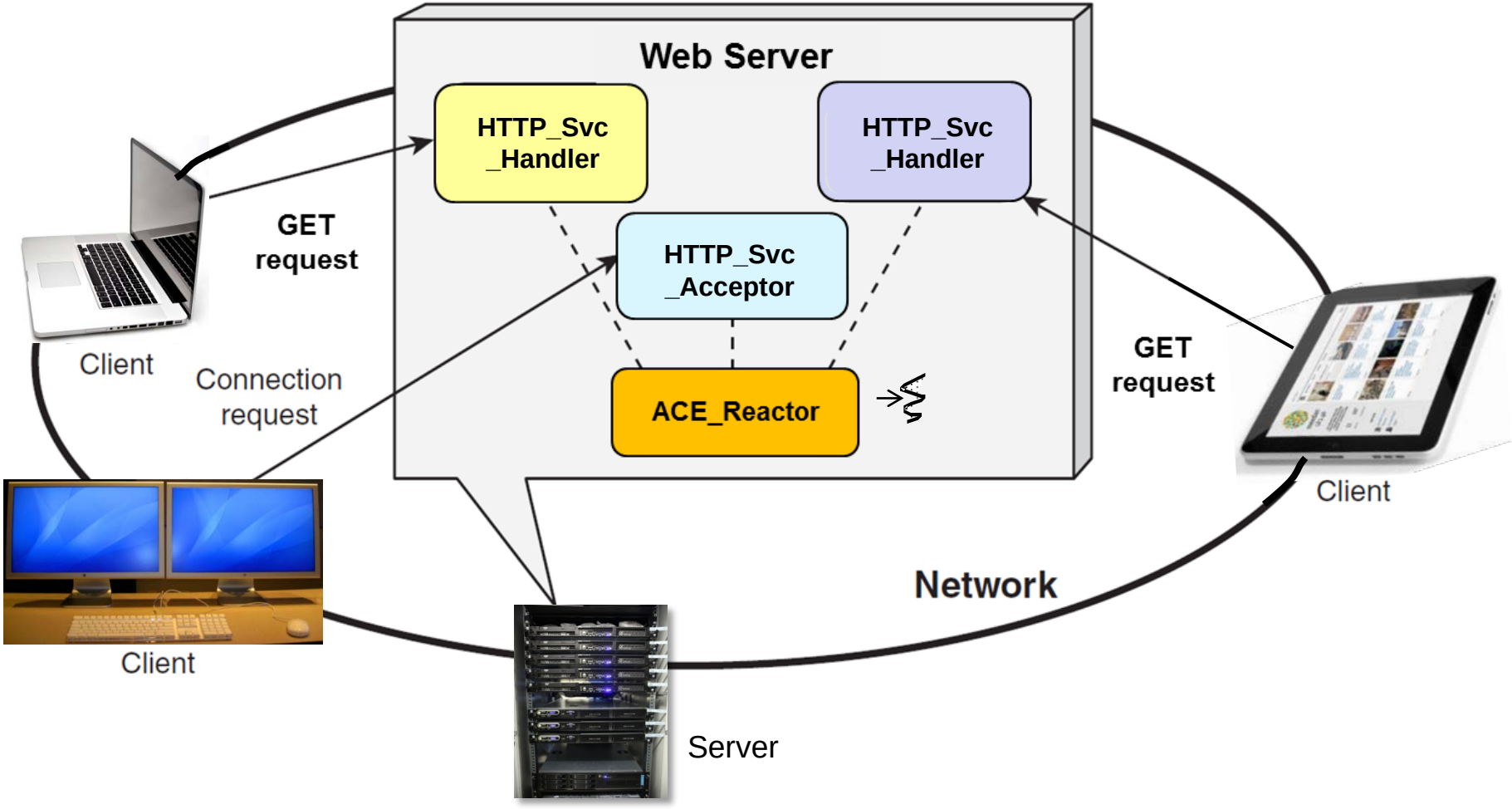
Note how ***handle_input()*** *implements the GoF Template Method pattern!*

ACE Acceptor-Connector Implementation of JAWS



Reactive Event Handling with ACE_Acceptor

This example shows how to use **`ACE_Acceptor`** & **`ACE_Svc_Handler`** to implement a reactive web server



There's *very* little app code since the ACE frameworks do much of the work!

Reactive Event Handling with ACE_Svc_Handler

```
class HTTP_Svc_Handler
  : public ACE_Svc_Handler<ACE_SOCK_Stream, ACE_NULL_SYNCH> {

public:
  ...

  virtual int handle_input(ACE_HANDLE) {

    std::string pathname (get_pathname (peer ()));
    ACE_Mem_Map mapped_file (pathname.c_str ());

    return peer.send_n (mapped_file.addr (),
                        mapped_file.size ());
  }
...
```

Be a service handler that uses `ACE_SOCK_Stream` to communicate with clients

Called back by reactor when a GET request arrives

Extract path from GET request

Memory map the requested content

Transmits content back to client

This “business logic” is the main piece of code you need to write!

Reactive Event Handling with ACE_Acceptor

```
1 typedef ACE_Acceptor<HTTP_Svc_Handler, ACE_SOCK_Acceptor>
2         HTTP_Svc_Acceptor;
3 typedef Reactor_HTTP_Server<HTTP_Svc_Acceptor>
4         HTTP_Server_Daemon;

5 int main (int argc, char *argv[]) {
6   ACE_Reactor reactor;

7   new HTTP_Server_Daemon (argc, argv, &reactor);
8

9   reactor.run_reactor_event_loop ();
    ...
10 }
```

Instantiate ACE_Acceptor

Reuse our earlier template driver & Instantiate it with the HTTP_Svc_Acceptor

Event loop controller object

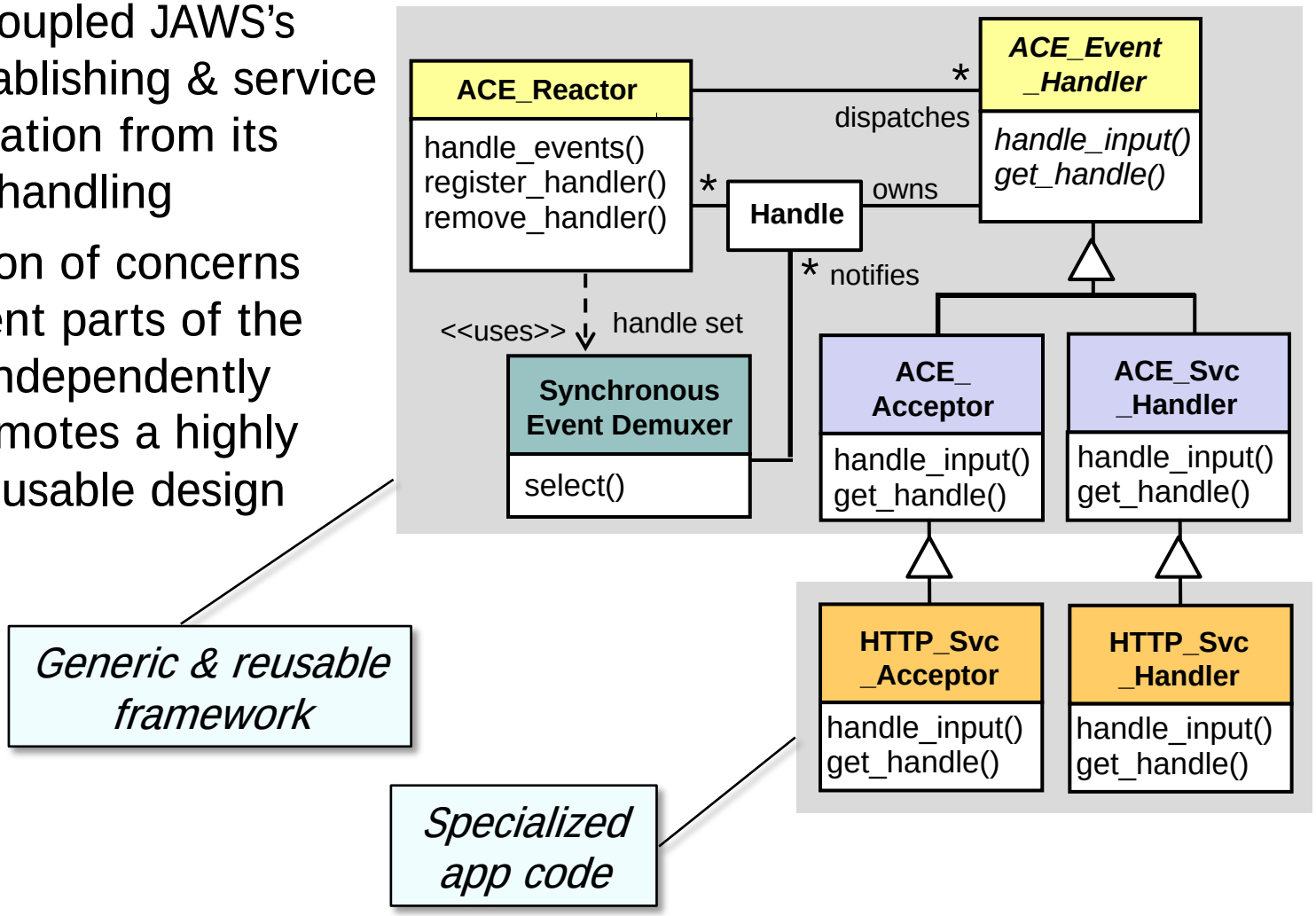
Dynamic allocation ensures proper deletion semantics

Run server event loop

Note the extensive design & code reuse here!

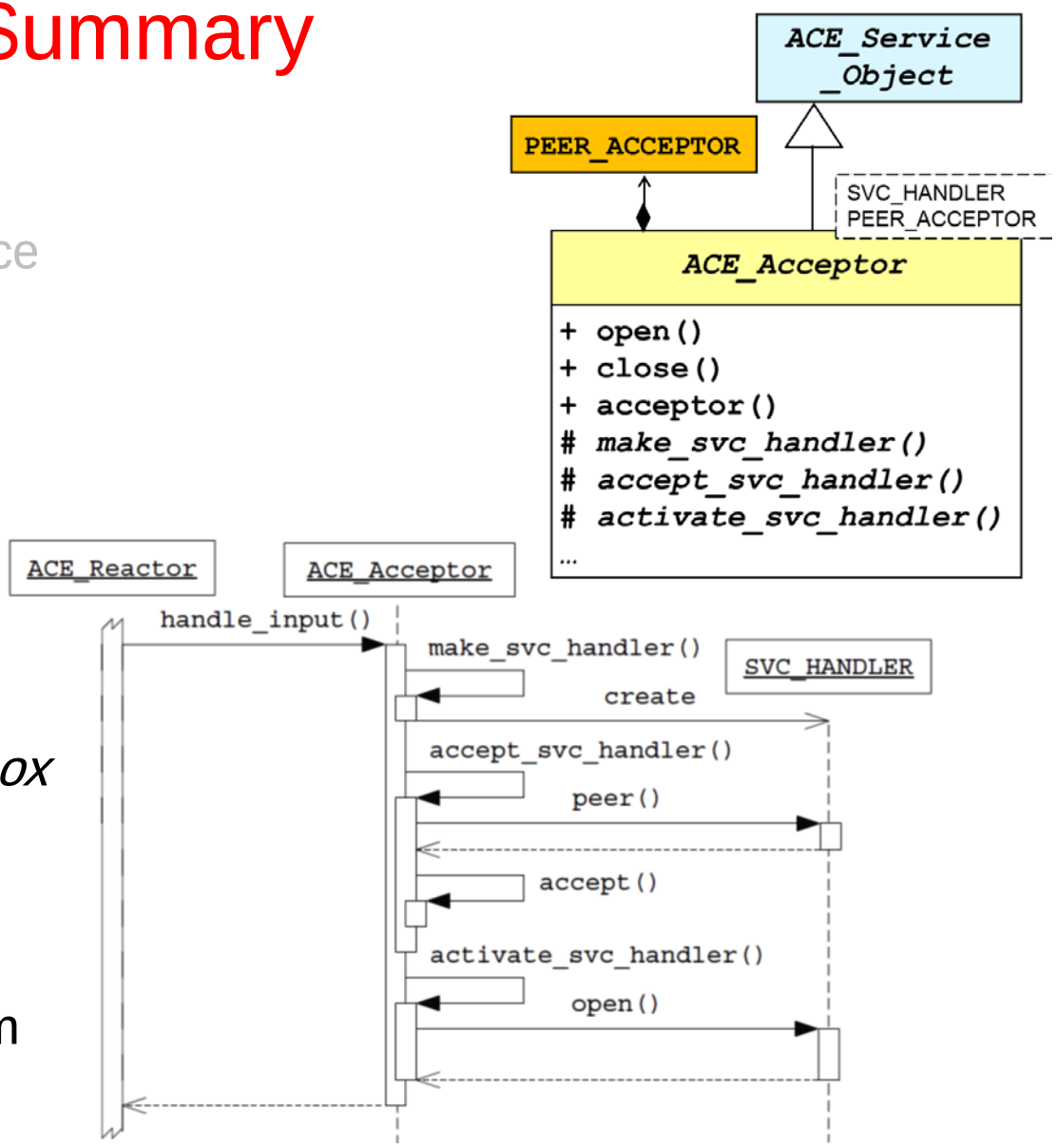
Summary

- The ACE *Acceptor-Connector* framework decoupled JAWS's connection establishing & service handler initialization from its HTTP protocol handling
  - This separation of concerns allows different parts of the software to independently evolve & promotes a highly modular & reusable design



Summary

- The ACE *Acceptor-Connector* framework decoupled JAWS's connection establishing & service handler initialization from its HTTP protocol handling
  - This separation of concerns allows different parts of the software to independently evolve & promotes a highly modular & reusable design
- Combining *white-box* & *black-box* reuse helps minimize code written by app developers
  - Connection establishment & service initialization code from reactive web server is gone!



At the heart of these framework designs is *commonality* & *variability* analysis