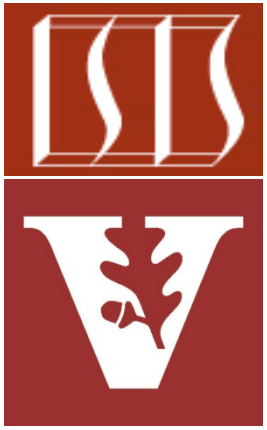
Patterns & Frameworks for Service Configuration & Activation: Part 1

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



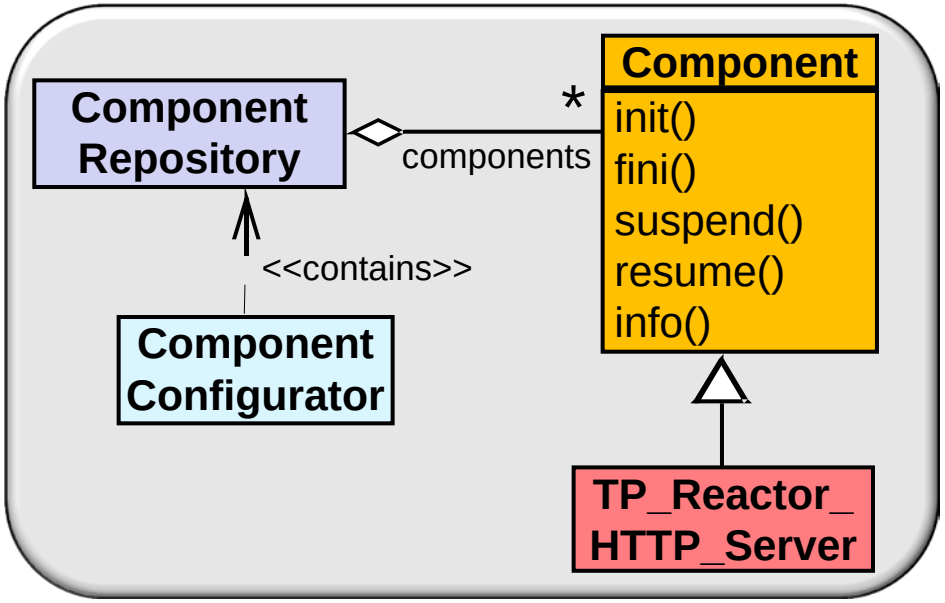
Professor of Computer Science

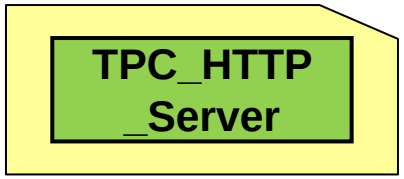
Institute for Software Integrated Systems

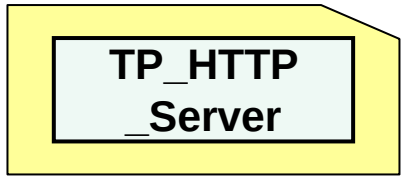
Vanderbilt University
Nashville, Tennessee, USA

Topics Covered in this Part of the Module

- Describe the *Component Configurator* pattern

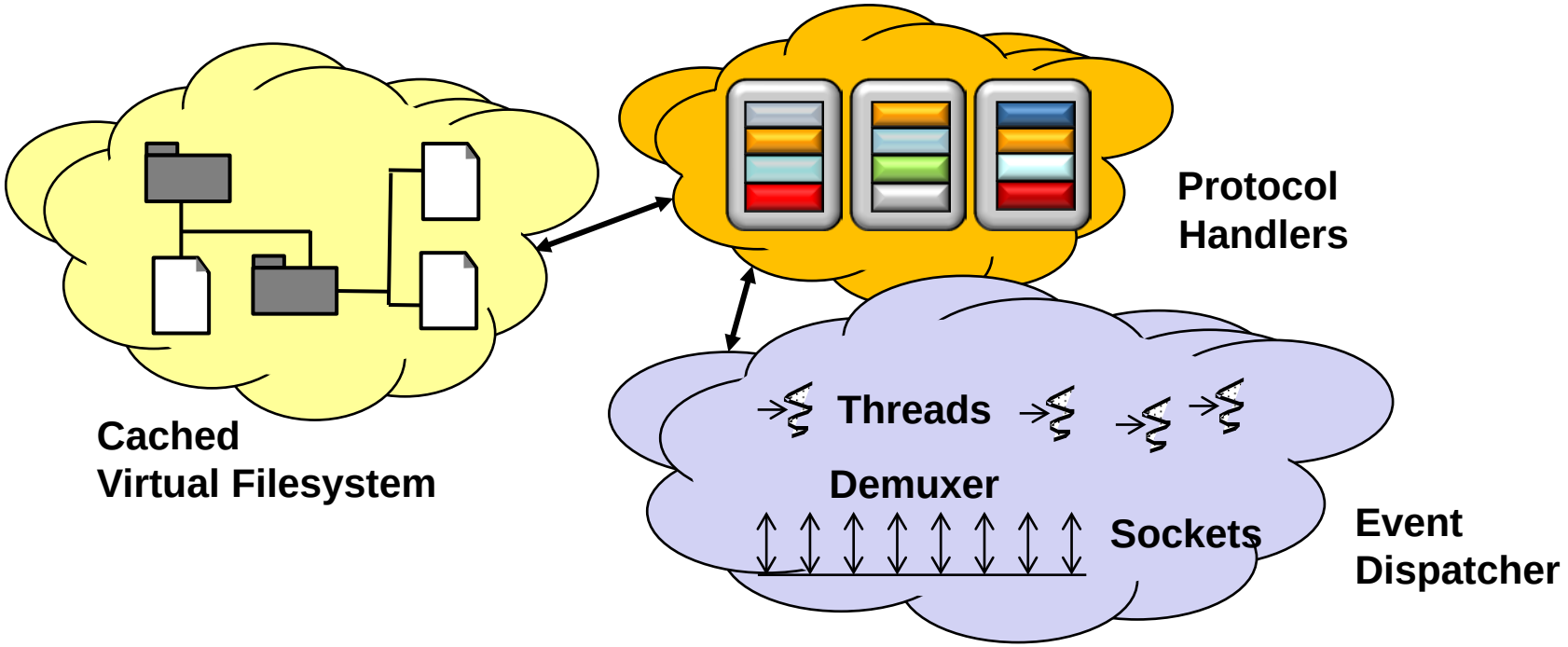






Enhancing Server (Re)Configurability

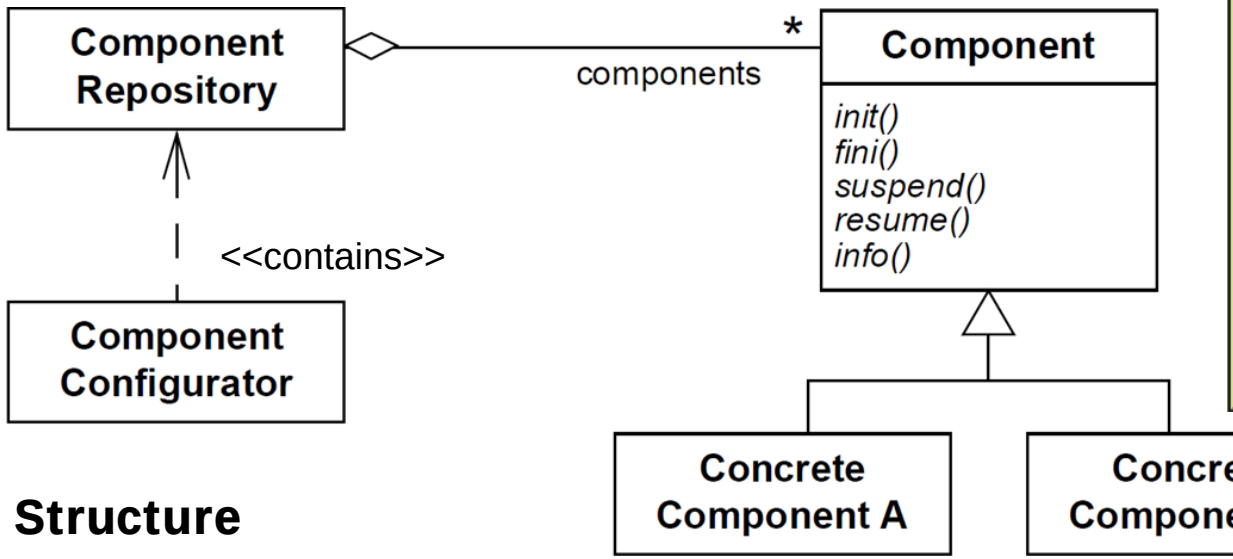
Context	Problem
• Some implementations of web server components depend on static or dynamic factors • e.g., # of cores, version of the OS, system workload, etc.	• Prematurely committing to a web server configuration is inflexible/inefficient since some decisions can't be made at design-time & apps incur overhead for unused or unneeded components



Enhancing Server (Re)Configurability

Context	Problem	Solution

• Some implementations of web server components depend on static or dynamic factors
  • e.g., # of cores, version of the OS, system workload, etc.
	• Prematurely committing to a web server configuration is inflexible/inefficient since some decisions can't be made at design-time & apps incur overhead for unused or unneeded components	• Apply the ***Component Configurator*** pattern to assemble desired web server components dynamically
		  • e.g., at installation-time or at runtime



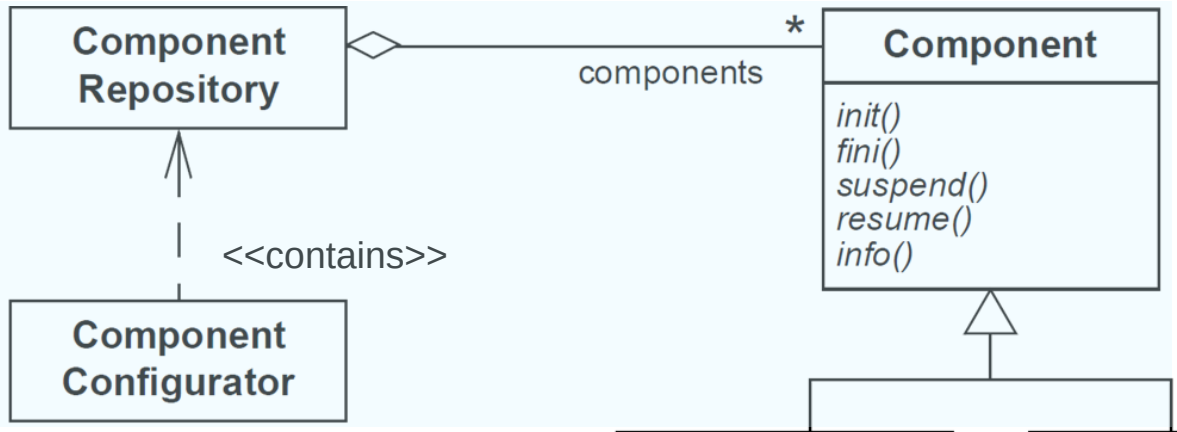
Structure

Component Configurator decouples component interfaces from their implementations & can (re)configure components without having to shutdown & restart running apps

Enhancing Server (Re)Configurability

Context	Problem	Solution

• Some implementations of web server components depend on static or dynamic factors
  • e.g., # of cores, version of the OS, system workload, etc.
	• Prematurely committing to a web server configuration is inflexible/inefficient since some decisions can't be made at design-time & apps incur overhead for unused or unneeded components	• Apply the ***Component Configurator*** pattern to assemble desired web server components dynamically
		  • e.g., at installation-time or at runtime



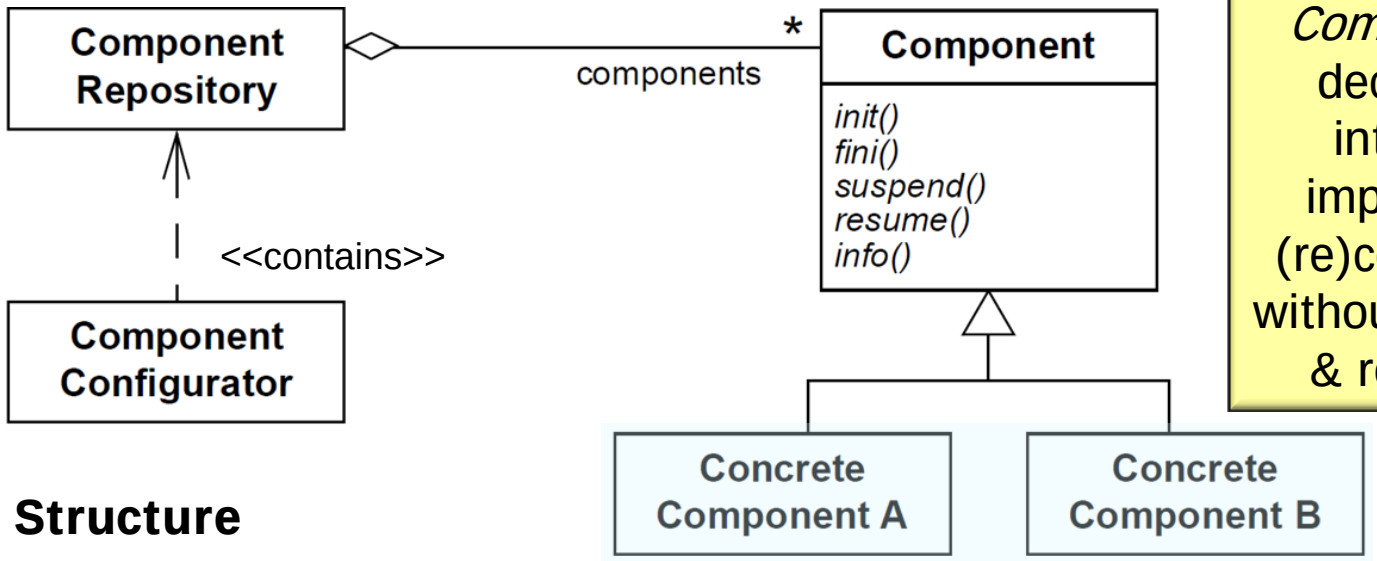
Structure

Component Configurator decouples component interfaces from their implementations & can (re)configure components without having to shutdown & restart running apps

Enhancing Server (Re)Configurability

Context	Problem	Solution

• Some implementations of web server components depend on static or dynamic factors
  • e.g., # of cores, version of the OS, system workload, etc.
	• Prematurely committing to a web server configuration is inflexible/inefficient since some decisions can't be made at design-time & apps incur overhead for unused or unneeded components	• Apply the ***Component Configurator*** pattern to assemble desired web server components dynamically
		  • e.g., at installation-time or at runtime



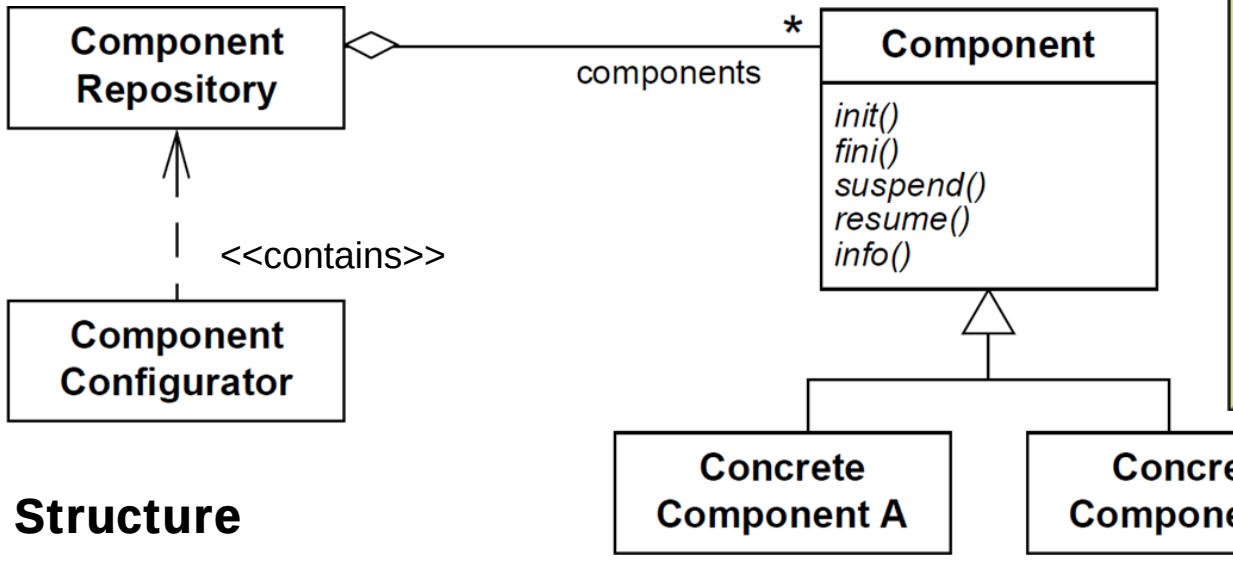
Structure

Component Configurator decouples component interfaces from their implementations & can (re)configure components without having to shutdown & restart running apps

Enhancing Server (Re)Configurability

Context	Problem	Solution

• Some implementations of web server components depend on static or dynamic factors
  • e.g., # of cores, version of the OS, system workload, etc.
	• Prematurely committing to a web server configuration is inflexible/inefficient since some decisions can't be made at design-time & apps incur overhead for unused or unneeded components	• Apply the ***Component Configurator*** pattern to assemble desired web server components dynamically
		  • e.g., at installation-time or at runtime



Structure

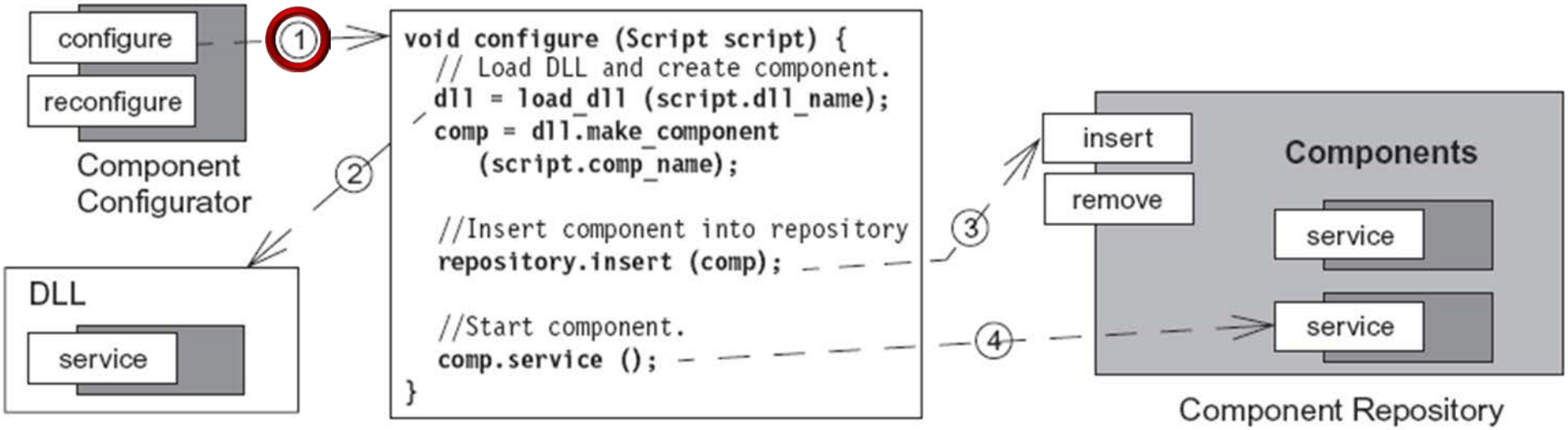
Component Configurator decouples component interfaces from their implementations & can (re)configure components without having to shutdown & restart running apps

See www.dre.vanderbilt.edu/~schmidt/PDF/Svc-Conf.pdf for more info

Enhancing Server (Re)Configurability

Context	Problem	Solution

• Some implementations of web server components depend on static or dynamic factors
  • e.g., # of cores, version of the OS, system workload, etc.
	• Prematurely committing to a web server configuration is inflexible/inefficient since some decisions can't be made at design-time & apps incur overhead for unused or unneeded components	• Apply the ***Component Configurator*** pattern to assemble desired web server components dynamically
		  • e.g., at installation-time or at runtime

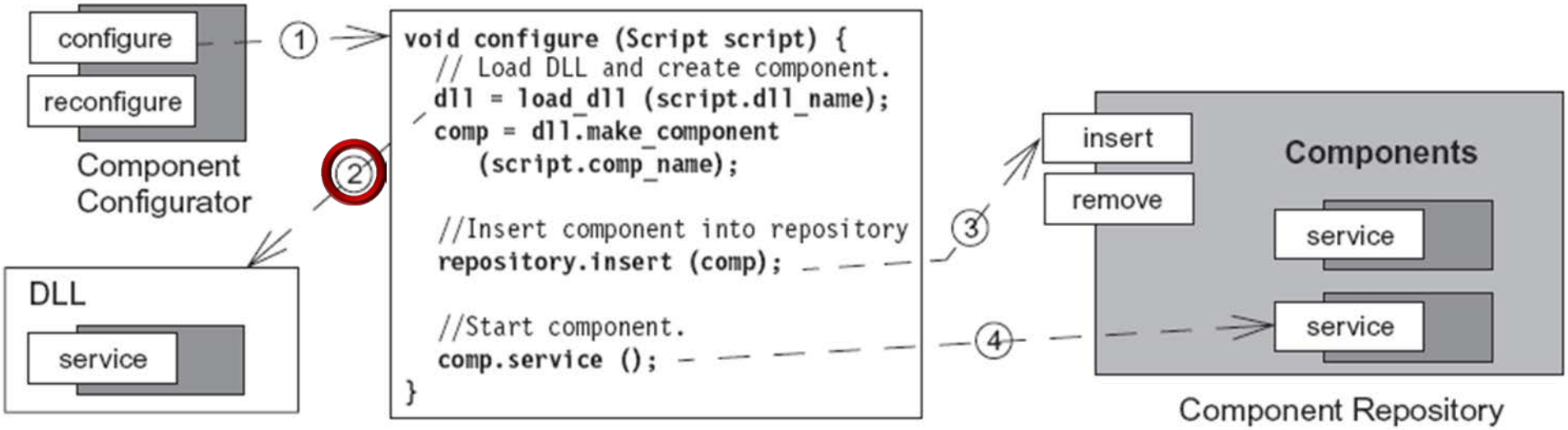


Dynamics

Enhancing Server (Re)Configurability

Context	Problem	Solution

• Some implementations of web server components depend on static or dynamic factors
  • e.g., # of cores, version of the OS, system workload, etc.
	• Prematurely committing to a web server configuration is inflexible/inefficient since some decisions can't be made at design-time & apps incur overhead for unused or unneeded components	• Apply the ***Component Configurator*** pattern to assemble desired web server components dynamically
		  • e.g., at installation-time or at runtime

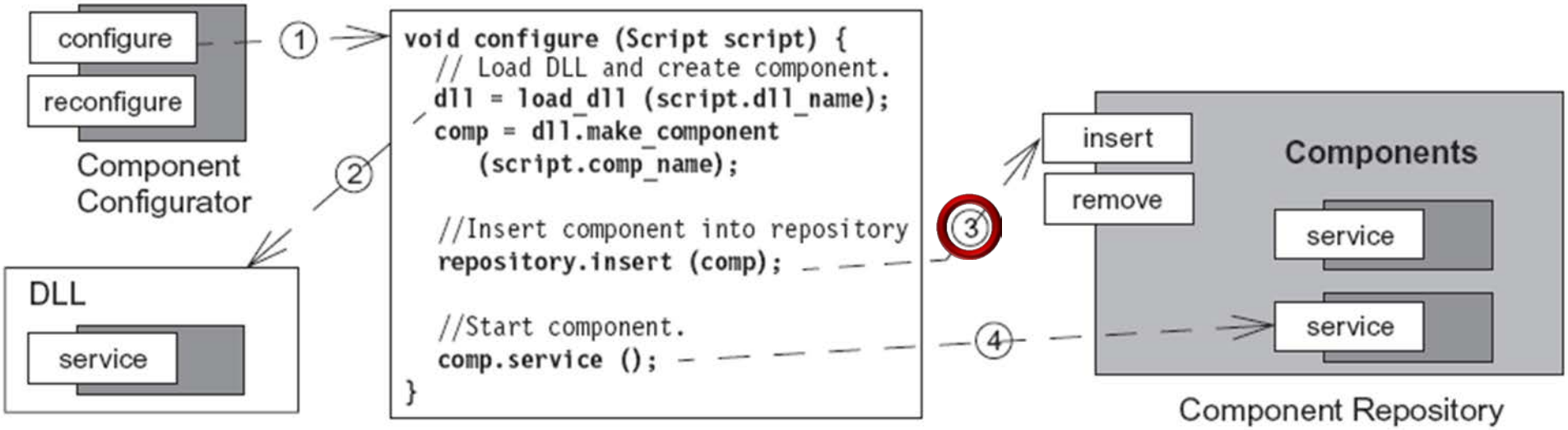


Dynamics

Enhancing Server (Re)Configurability

Context	Problem	Solution

• Some implementations of web server components depend on static or dynamic factors
  • e.g., # of cores, version of the OS, system workload, etc.
	• Prematurely committing to a web server configuration is inflexible/inefficient since some decisions can't be made at design-time & apps incur overhead for unused or unneeded components	• Apply the ***Component Configurator*** pattern to assemble desired web server components dynamically
		  • e.g., at installation-time or at runtime

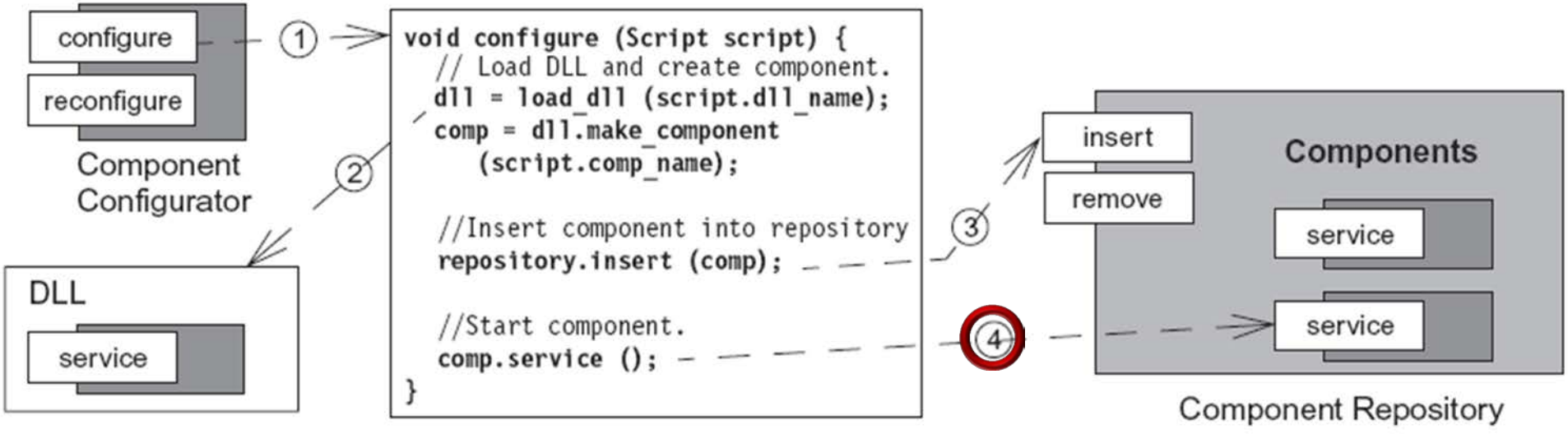


Dynamics

Enhancing Server (Re)Configurability

Context	Problem	Solution

• Some implementations of web server components depend on static or dynamic factors
  • e.g., # of cores, version of the OS, system workload, etc.
	• Prematurely committing to a web server configuration is inflexible/inefficient since some decisions can't be made at design-time & apps incur overhead for unused or unneeded components	• Apply the ***Component Configurator*** pattern to assemble desired web server components dynamically
		  • e.g., at installation-time or at runtime

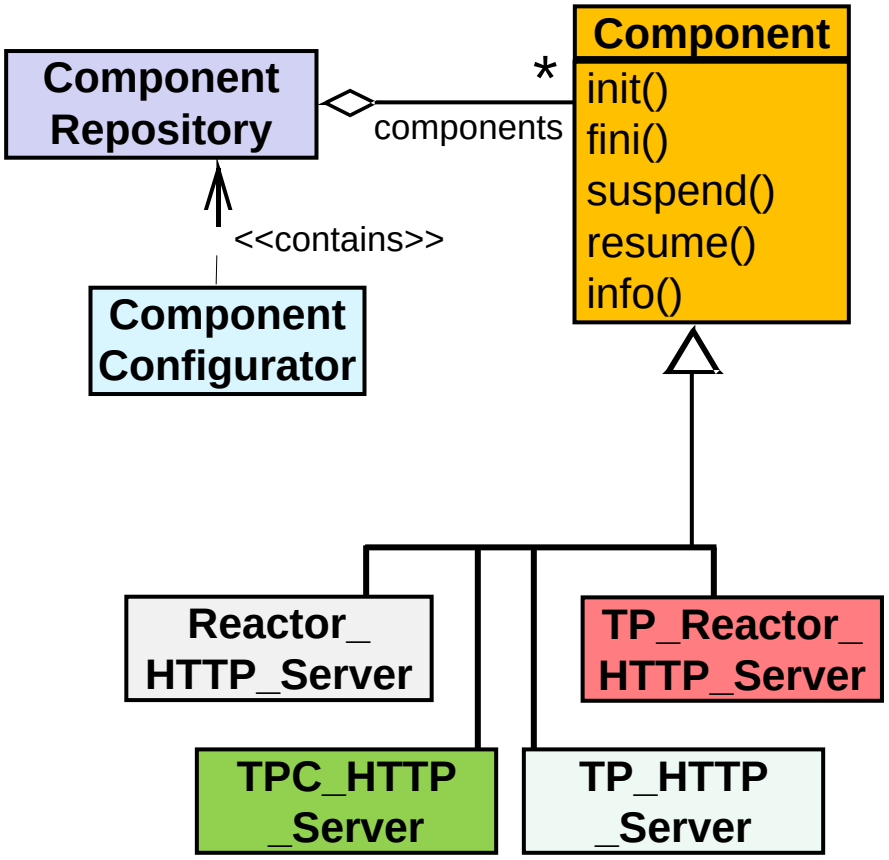


Dynamics

Examples include loadable device drivers, Java applets, & app servers

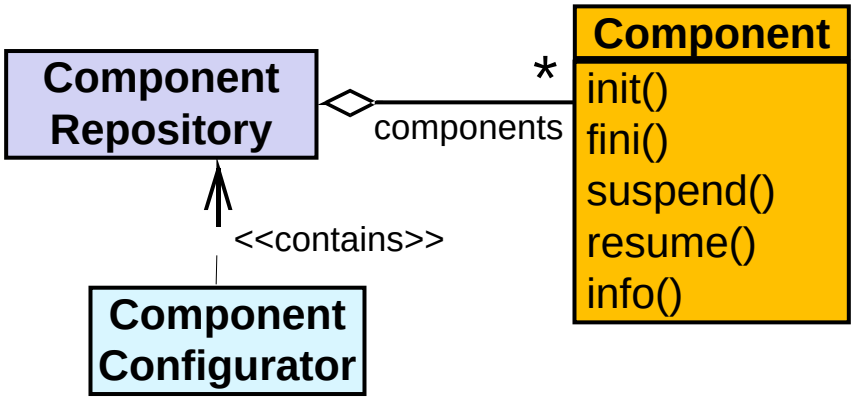
Applying Component Configurator to JAWS

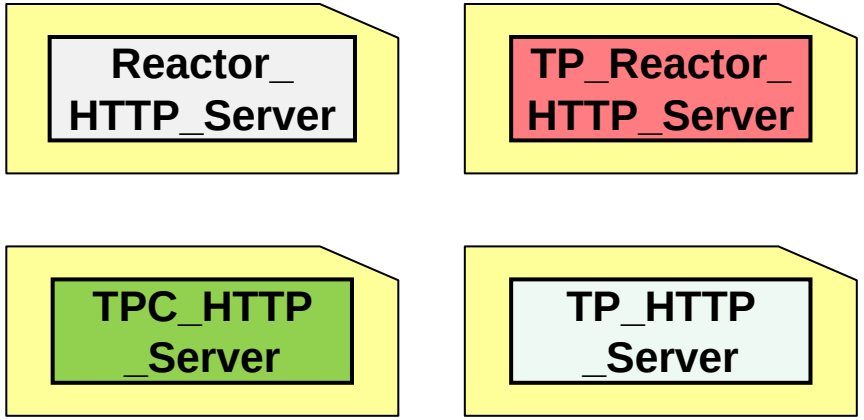
- JAWS can apply the *Component Configurator* pattern to dynamically assemble various configurations
  - e.g., different threading models, different termination schemes, etc.



Applying Component Configurator to JAWS

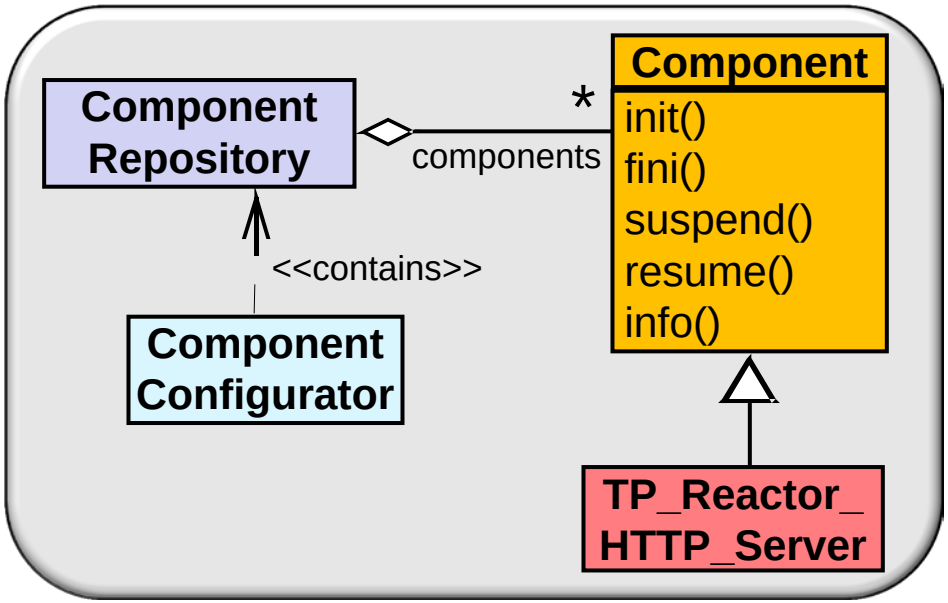
- JAWS can apply the *Component Configurator* pattern to dynamically assemble various configurations
  - e.g., different threading models, different termination schemes, etc.
- Concrete components can be packaged into a suitable unit of configuration
  - e.g., as a dynamically linked library (DLL) or shared library

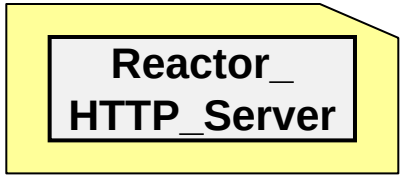


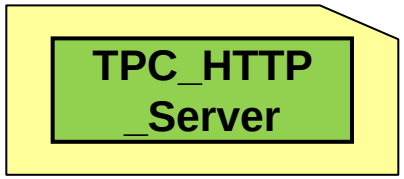


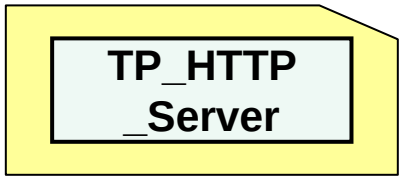
Applying Component Configurator to JAWS

- JAWS can apply the *Component Configurator* pattern to dynamically assemble various configurations
  - e.g., different threading models, different termination schemes, etc.
- Concrete components can be packaged into a suitable unit of configuration
  - e.g., as a dynamically linked library (DLL) or shared library
- Only components that are currently in use are configured into a JAWS web server process
  - Reduces memory footprint





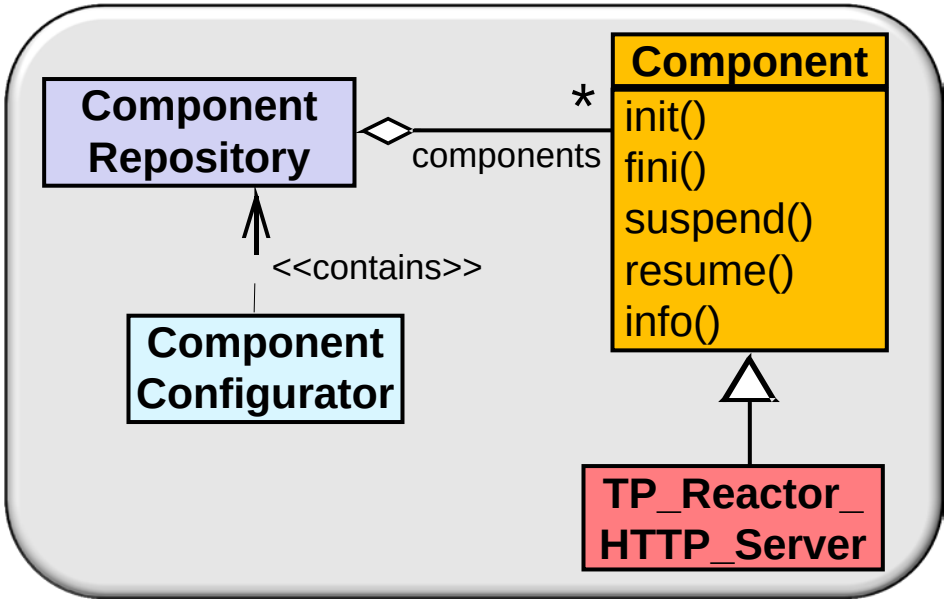


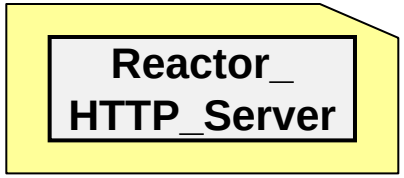


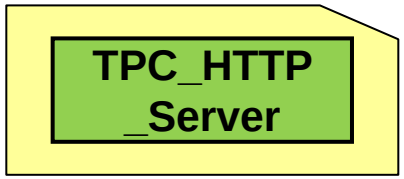
Benefits of Component Configurator

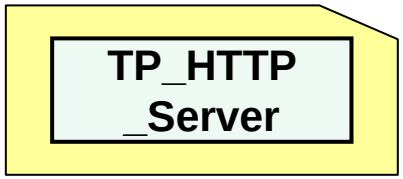
Parsimony

- Only incur memory overhead for components that are actually used









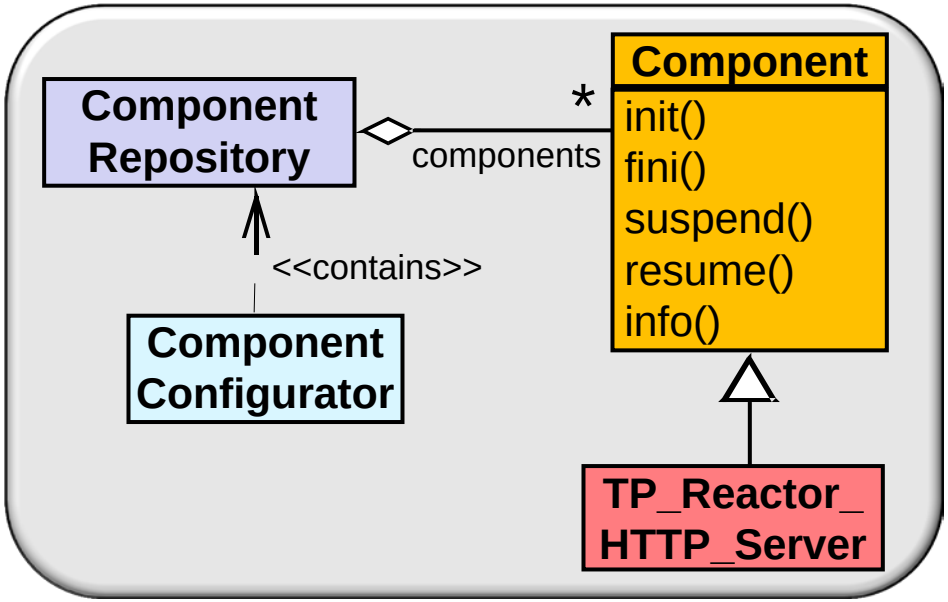
Benefits of Component Configurator

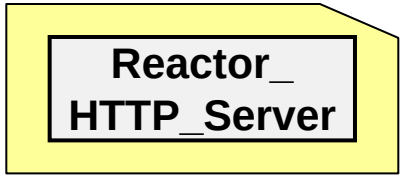
Parsimony

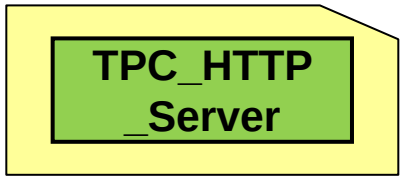
- Only incur memory overhead for components that are actually used

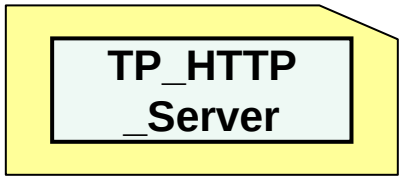
Centralized administration

- Centralizes uniform initialization & termination activities









Benefits of Component Configurator

Parsimony

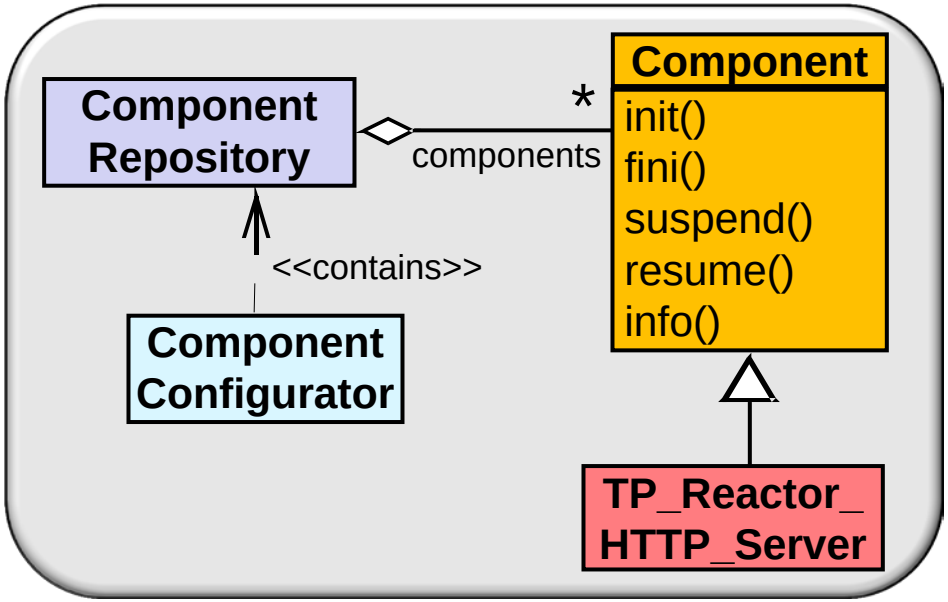
- Only incur memory overhead for components that are actually used

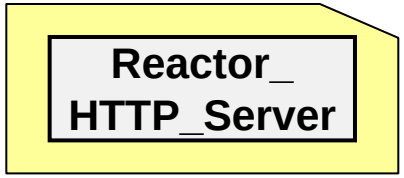
Centralized administration

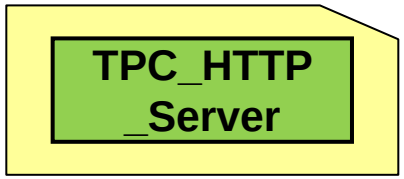
- Centralizes uniform initialization & termination activities

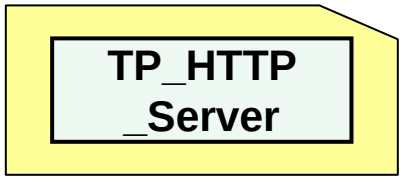
Modularity & testability

- Decouples components from manner in which they are configured into processes









Benefits of Component Configurator

Parsimony

- Only incur memory overhead for components that are actually used

Centralized administration

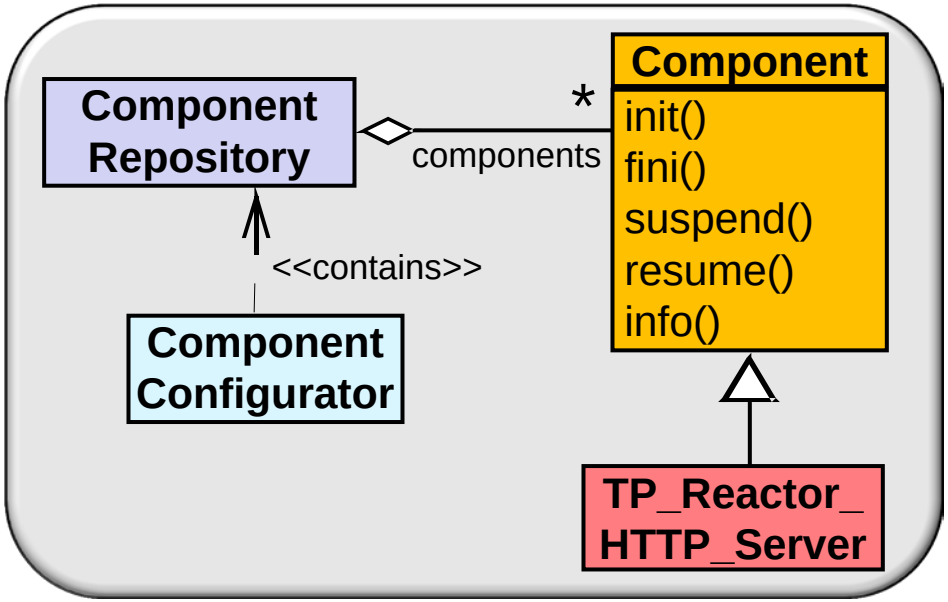
- Centralizes uniform initialization & termination activities

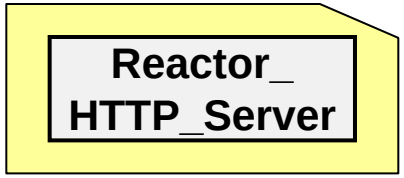
Modularity & testability

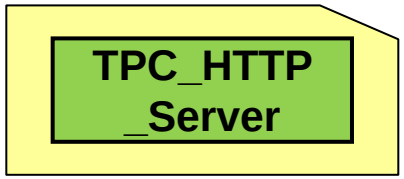
- Decouples components from manner in which they are configured into processes

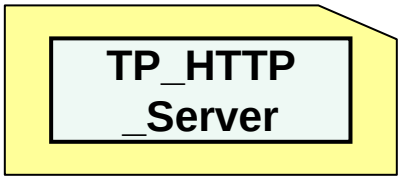
Configuration dynamism & control

- Can reconfigure components without modifying, recompiling, statically relinking existing code & without restarting the component or other collocated components





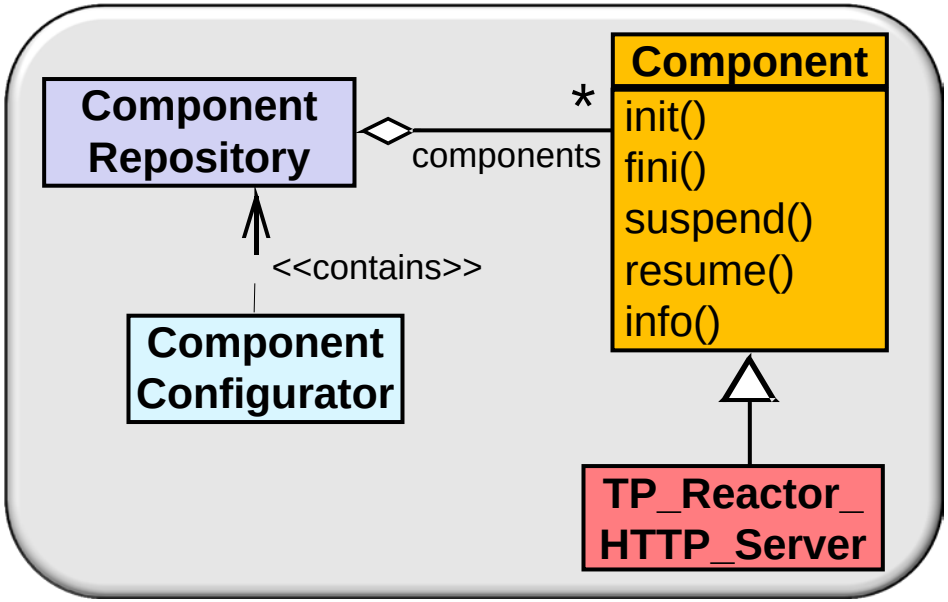


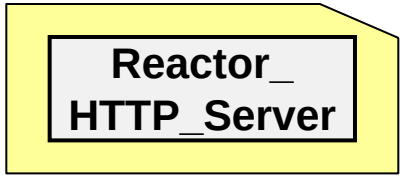


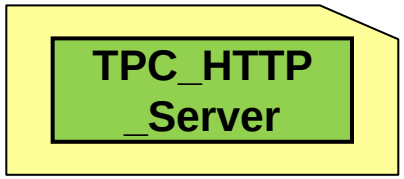
Limitations of Component Configurator

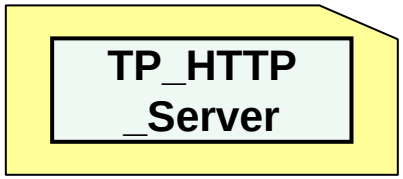
Lack of determinism & ordering dependencies

- Hard to determine/analyze behavior of app until its components are configured at run-time









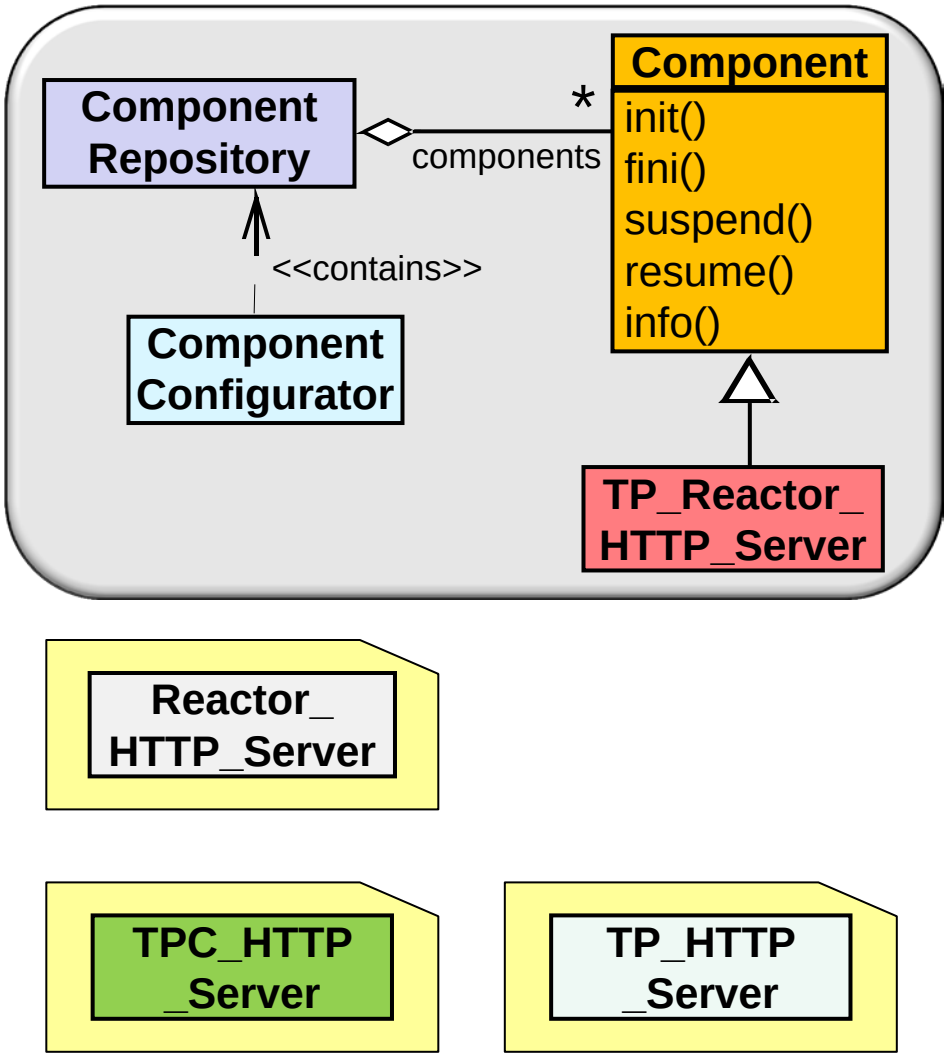
Limitations of Component Configurator

Lack of determinism & ordering dependencies

- Hard to determine/analyze behavior of app until its components are configured at run-time

Reduced security or reliability

- An app that uses this pattern may be less secure or reliable than an equivalent statically-configured app



Limitations of Component Configurator

Lack of determinism & ordering dependencies

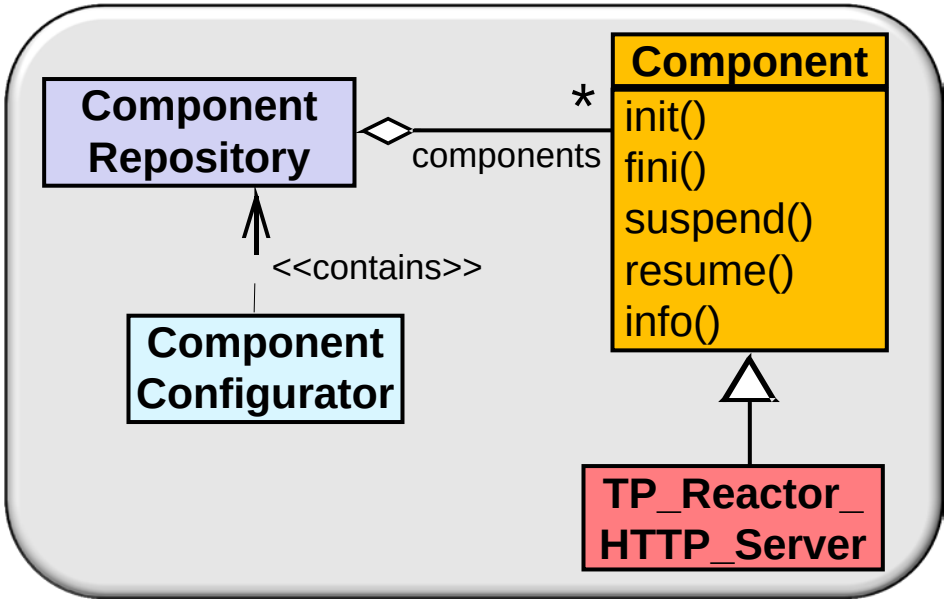
- Hard to determine/analyze behavior of app until its components are configured at run-time

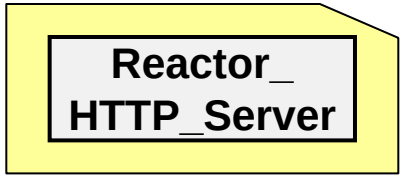
Reduced security or reliability

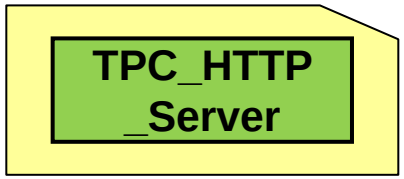
- An app that uses this pattern may be less secure or reliable than an equivalent statically-configured app

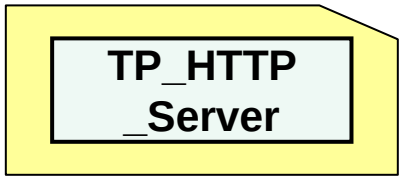
Increased run-time overhead & infrastructure complexity

- Additional abstraction & indirection when executing components









Limitations of Component Configurator

Lack of determinism & ordering dependencies

- Hard to determine/analyze behavior of app until its components are configured at run-time

Reduced security or reliability

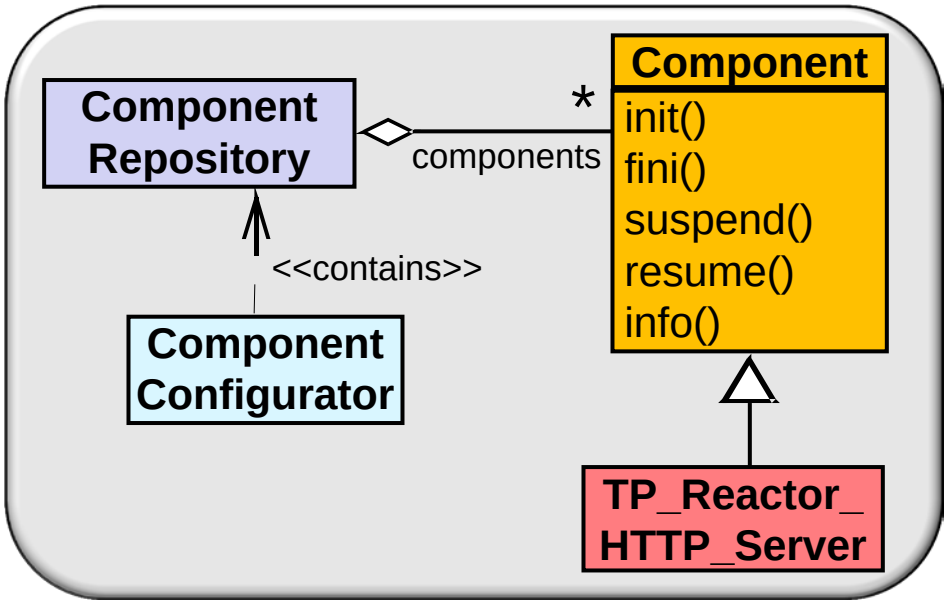
- An app that uses this pattern may be less secure or reliable than an equivalent statically-configured app

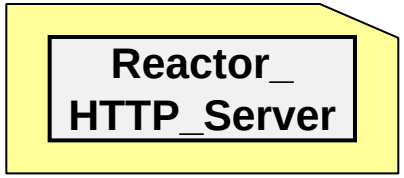
Increased run-time overhead & infrastructure complexity

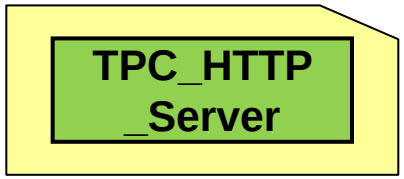
- Additional abstraction & indirection when executing components

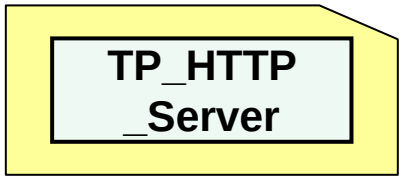
Overly narrow common interfaces

- Component initialization/termination may be too complicated or tightly coupled to perform uniformly



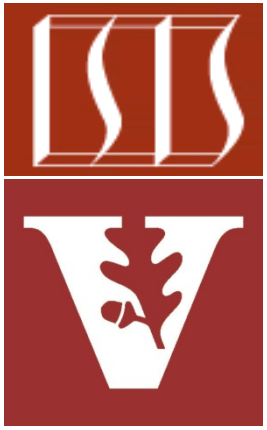






Patterns & Frameworks for Service Configuration & Activation: Part 2

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



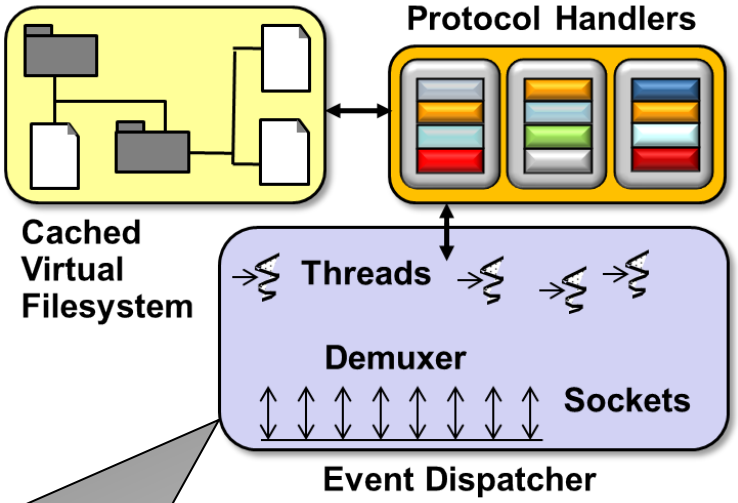
Professor of Computer Science

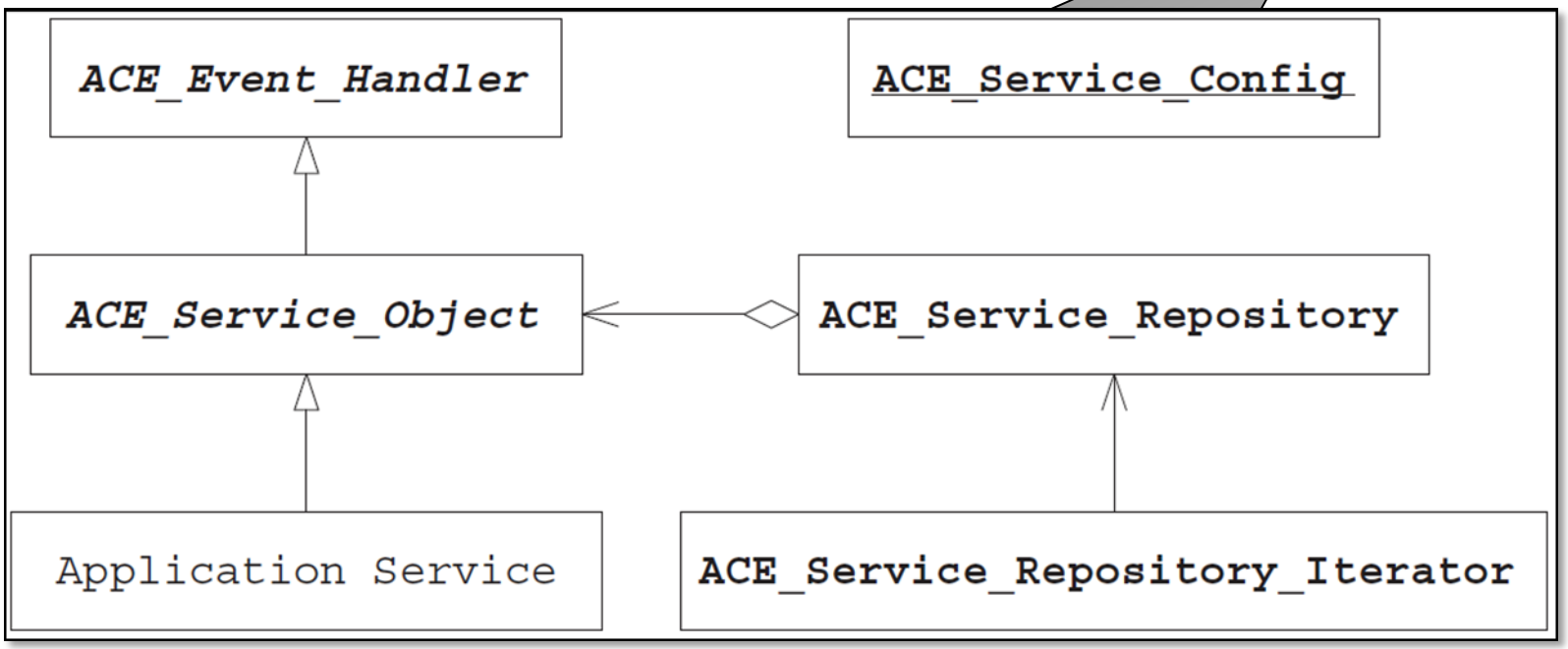
Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Topics Covered in this Part of the Module

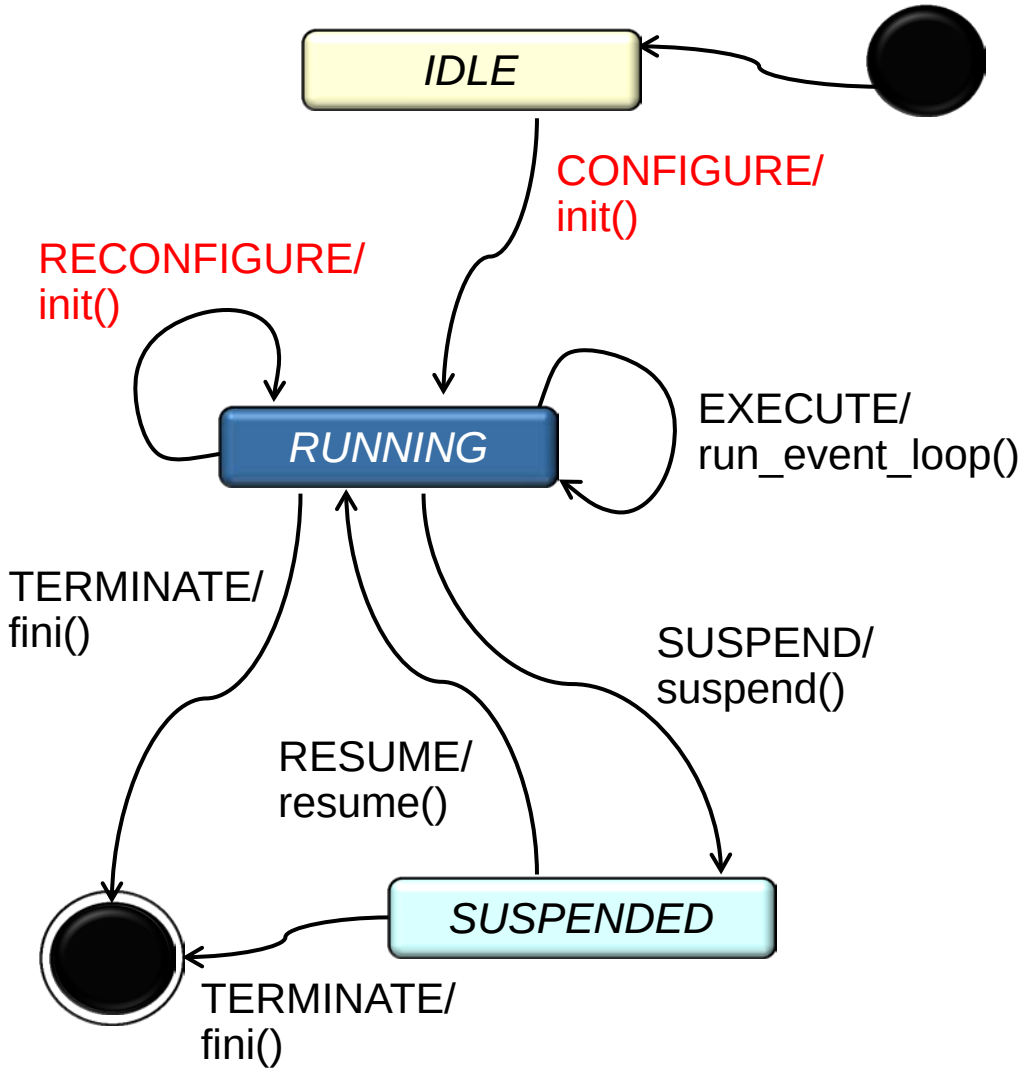
- Describe the *Component Configurator* pattern
- Describe the ACE *Service Configurator* framework





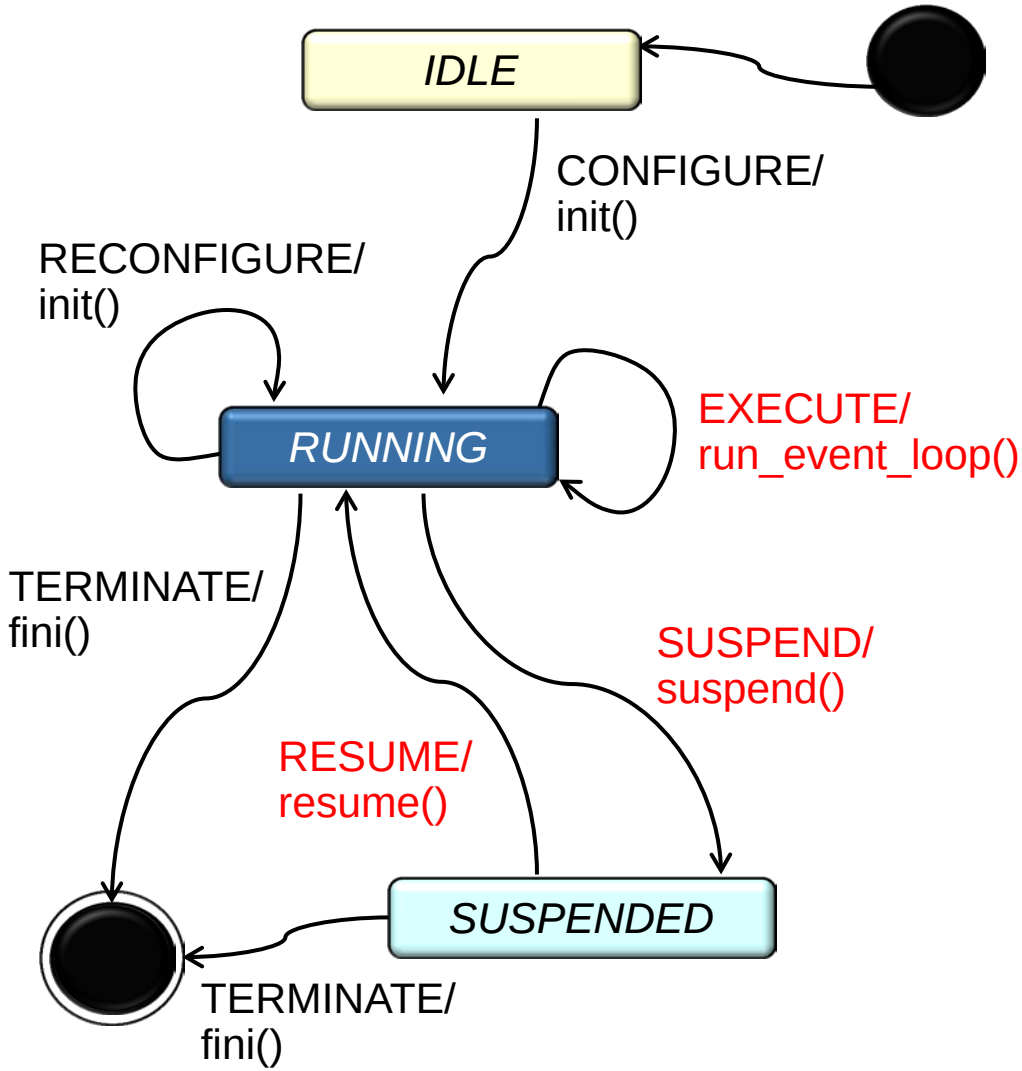
Motivation for the ACE Service Configurator Framework

- Configuring & managing service life cycles involves the following aspects:
  - Initialization



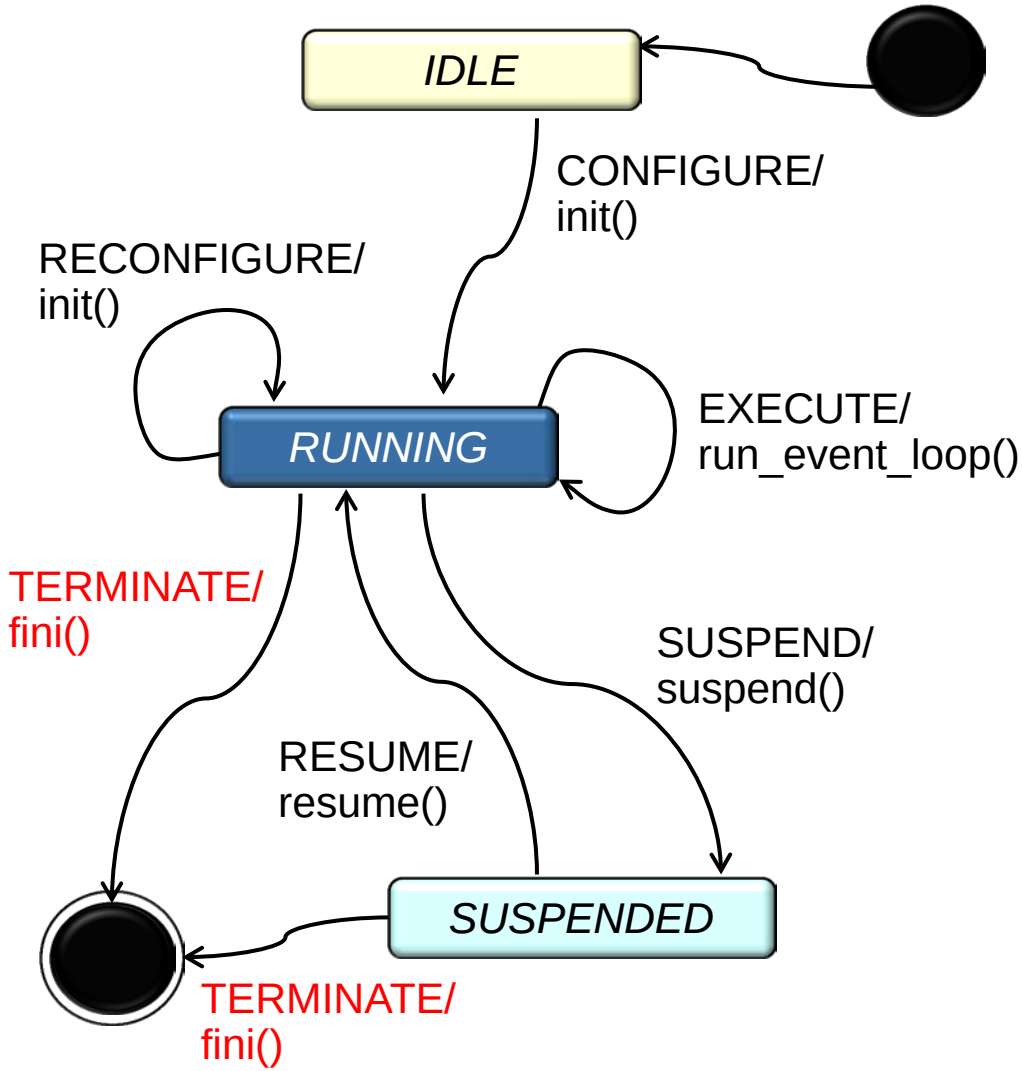
Motivation for the ACE Service Configurator Framework

- Configuring & managing service life cycles involves the following aspects:
  - Initialization
  - Execution control



Motivation for the ACE Service Configurator Framework

- Configuring & managing service life cycles involves the following aspects:
  - Initialization
  - Execution control
  - Termination

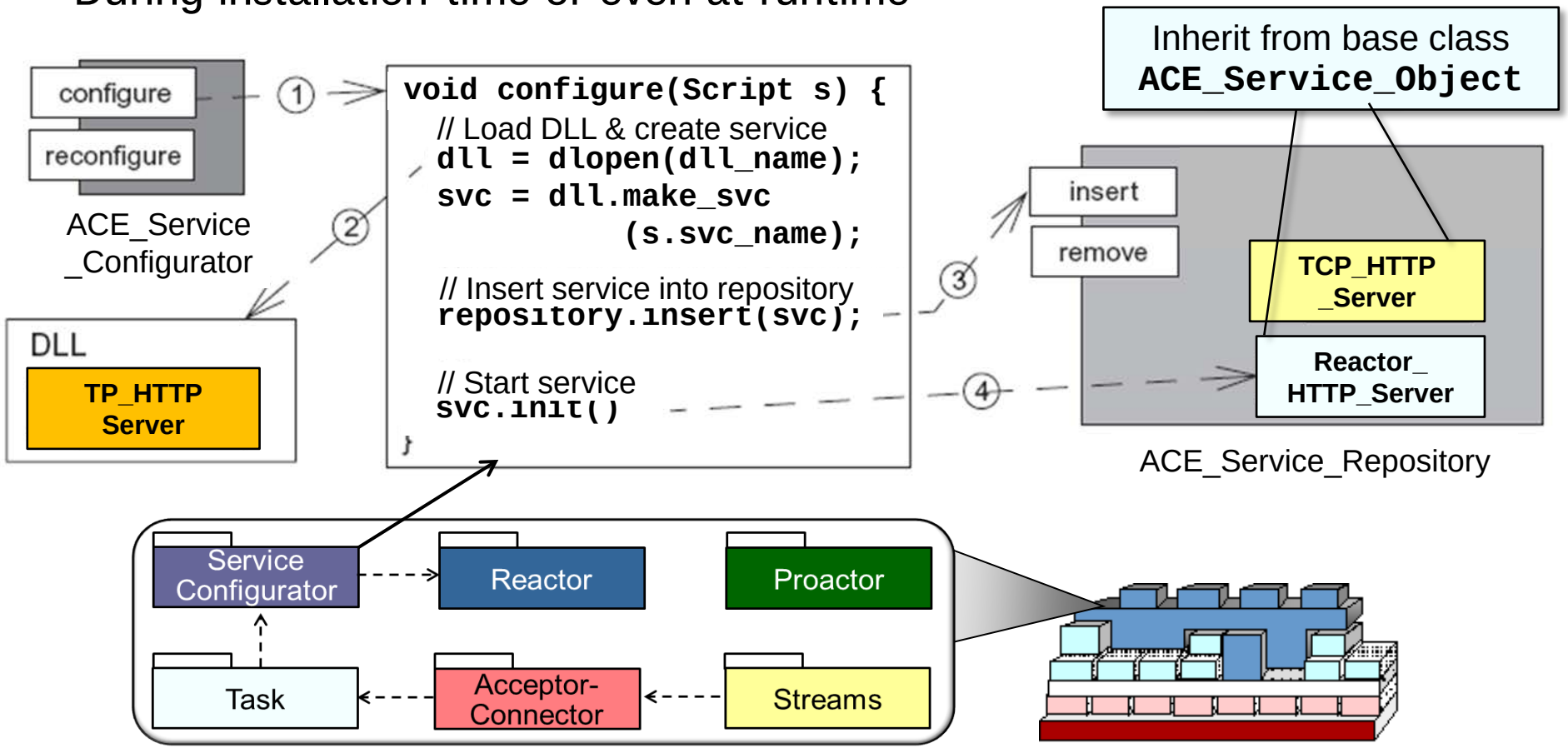


Motivation for the ACE Service Configurator Framework

- Configuring & managing service life cycles involves the following aspects:
  - Initialization
  - Execution control
  - Termination
- Developing these capabilities in an *ad hoc* manner can produce tightly coupled data structures & classes

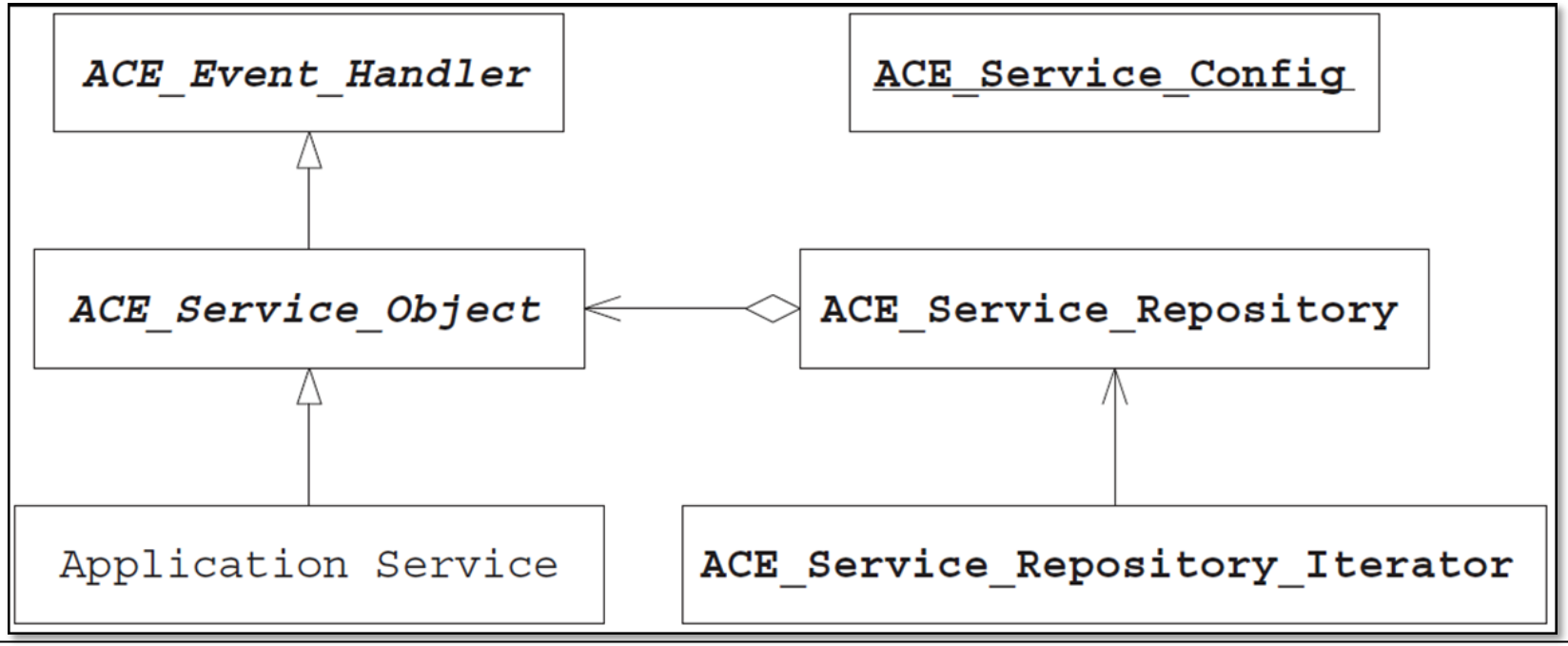
Overview of ACE Service Configurator Framework

- Classes in this framework allow apps to defer configuration & implementation decisions about their services until late in the design cycle
  - During installation-time or even at runtime



Overview of ACE Service Configurator Framework

- Classes in this framework allow apps to defer configuration & implementation decisions about their services until late in the design cycle
  - During installation-time or even at runtime
- Apps/services inherit from **`ACE_Service_Object`** & override its hook methods, which the ACE *Service Configurator* framework then dispatches to configure & control lifecycle of apps/service



Classes are designed in accordance with the *Component Configurator* pattern

Overview of ACE Service Configurator Framework

- Classes in this framework allow apps to defer configuration & implementation decisions about their services until late in the design cycle
  - During installation-time or even at runtime
- Apps/services inherit from **`ACE_Service_Object`** & override its hook methods, which the ACE *Service Configurator* framework then dispatches to configure & control lifecycle of apps/service
- Key classes in the ACE *Service Configurator* framework include

ACE Class	Description
`ACE_Service_Object`	Defines a uniform interface that ACE *Service Configurator* framework uses to configure & control a service implementation's lifecycle, including initializing, suspending, resuming, & terminating a service
`ACE_Service_Repository`	A central repository for all services managed by the ACE *Service Configurator* framework that provides methods for locating, reporting on, & controlling all app configured services
`ACE_Service_Config`	Provides an interpreter that parses & executes scripts specifying which services to (re)configure into an application (e.g., by linking & unlinking DLLs) & which services to suspend & resume

Overview of ACE Service Configurator Framework

- Classes in this framework allow apps to defer configuration & implementation decisions about their services until late in the design cycle
  - During installation-time or even at runtime
- Apps/services inherit from **`ACE_Service_Object`** & override its hook methods, which the ACE *Service Configurator* framework then dispatches to configure & control lifecycle of apps/service
- Key classes in the ACE *Service Configurator* framework include

ACE Class	Description
`ACE_Service_Object`	Defines a uniform interface that ACE *Service Configurator* framework uses to configure & control a service implementation's lifecycle, including initializing, suspending, resuming, & terminating a service
`ACE_Service_Repository`	A central repository for all services managed by the ACE *Service Configurator* framework that provides methods for locating, reporting on, & controlling all app configured services
`ACE_Service_Config`	Provides an interpreter that parses & executes scripts specifying which services to (re)configure into an application (e.g., by linking & unlinking DLLs) & which services to suspend & resume

Overview of ACE Service Configurator Framework

- Classes in this framework allow apps to defer configuration & implementation decisions about their services until late in the design cycle
  - During installation-time or even at runtime
- Apps/services inherit from **`ACE_Service_Object`** & override its hook methods, which the ACE *Service Configurator* framework then dispatches to configure & control lifecycle of apps/service
- Key classes in the ACE *Service Configurator* framework include

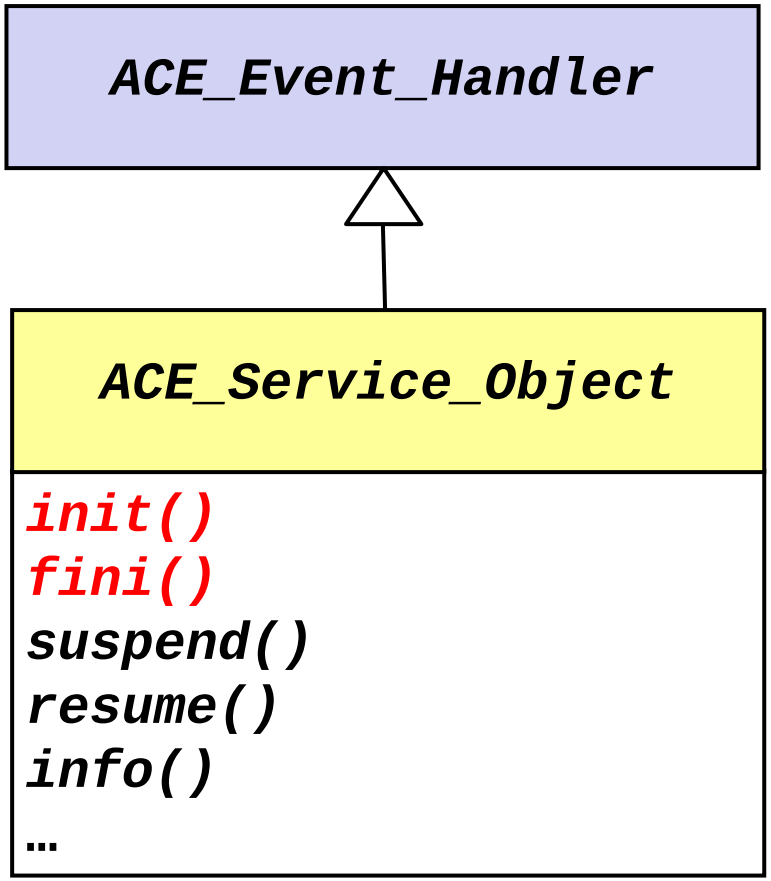
ACE Class	Description
`ACE_Service_Object`	Defines a uniform interface that ACE *Service Configurator* framework uses to configure & control a service implementation's lifecycle, including initializing, suspending, resuming, & terminating a service
`ACE_Service_Repository`	A central repository for all services managed by the ACE *Service Configurator* framework that provides methods for locating, reporting on, & controlling all app configured services
`ACE_Service_Config`	Provides an interpreter that parses & executes scripts specifying which services to (re)configure into an application (e.g., by linking & unlinking DLLs) & which services to suspend & resume

See www.dre.vanderbilt.edu/~schmidt/PDF/Svc-Conf.pdf for more info

The ACE_Service_Object Class

The base class that the ACE *Service Configurator* framework uses to configure & manage service implementations via the following hook methods:

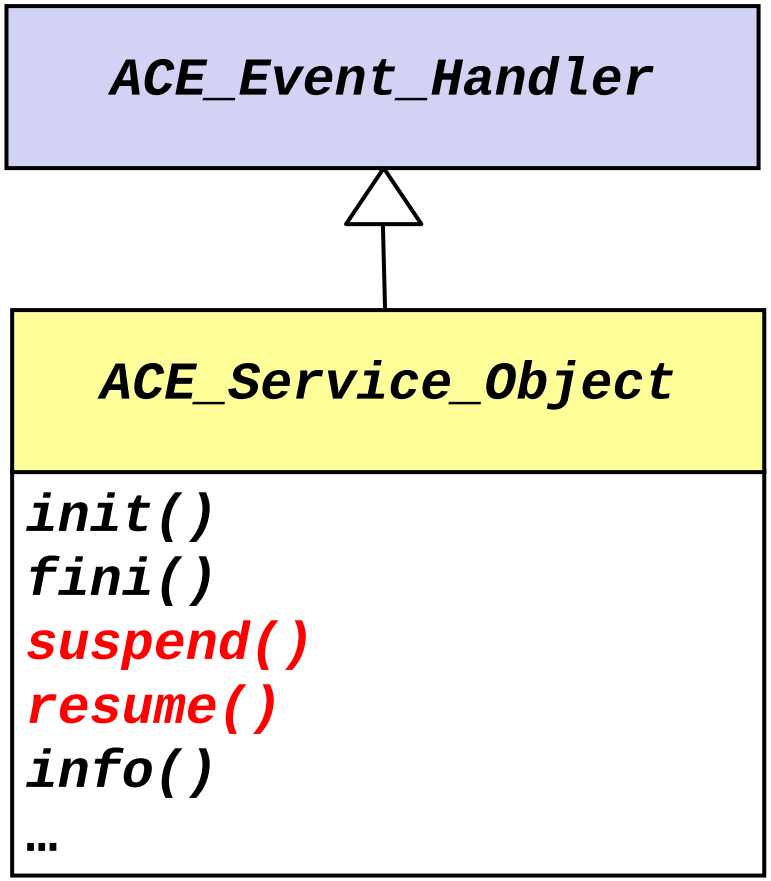
- Initialize & finalize a service



The ACE_Service_Object Class

The base class that the ACE *Service Configurator* framework uses to configure & manage service implementations via the following hook methods:

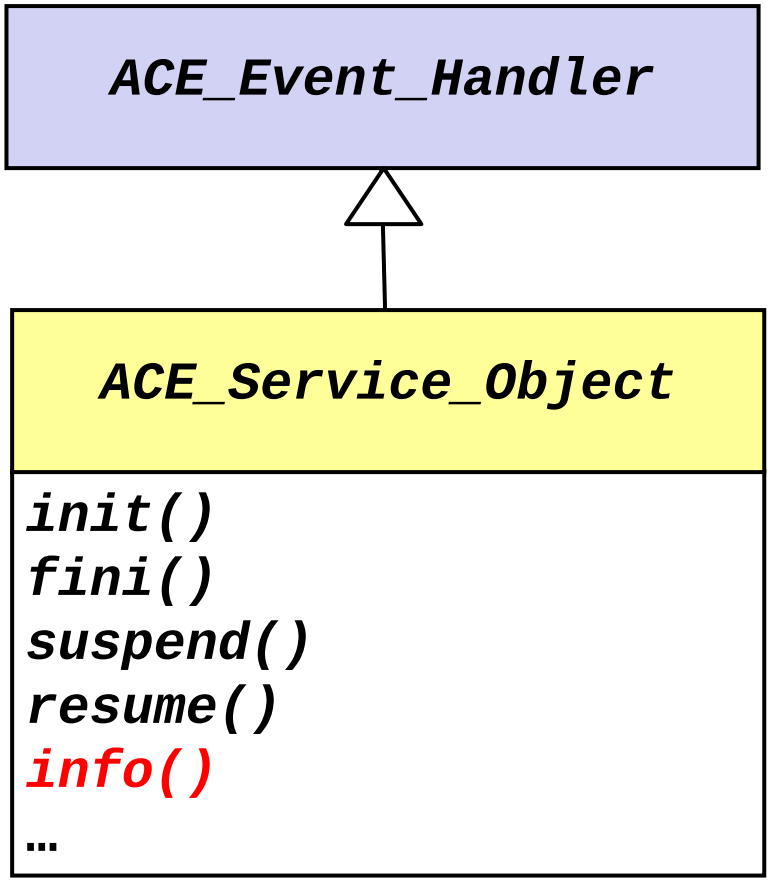
- Initialize & finalize a service
- Suspend service execution temporarily & resume execution of a suspended service



The ACE_Service_Object Class

The base class that the ACE *Service Configurator* framework uses to configure & manage service implementations via the following hook methods:

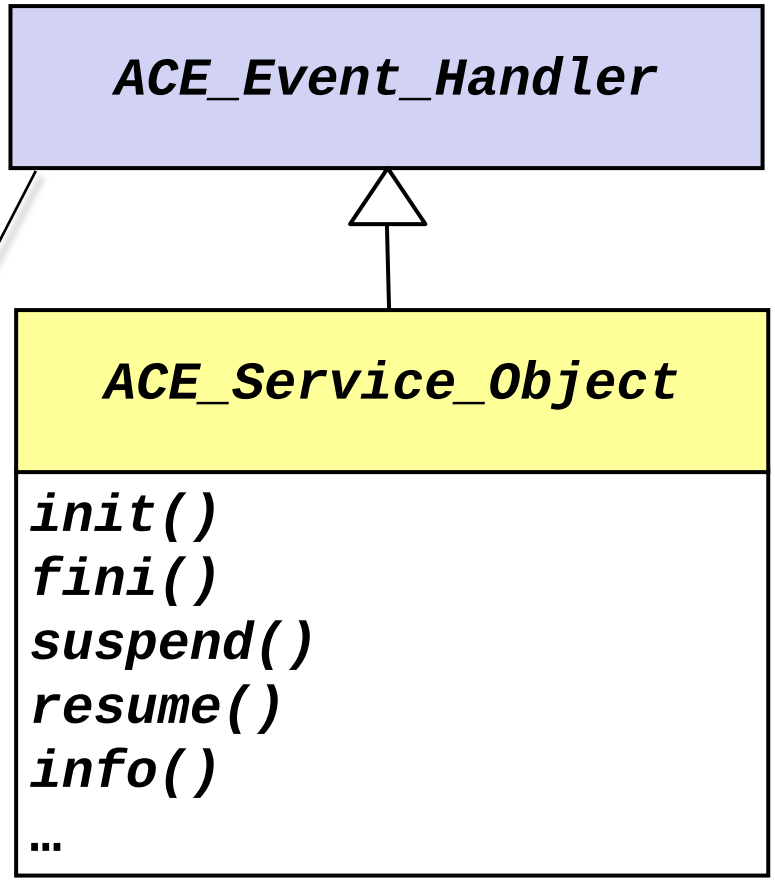
- Initialize & finalize a service
- Suspend service execution temporarily & resume execution of a suspended service
- Report key service information
  - e.g., its purpose, current status, port number where it listens for client connections, etc.



The ACE_Service_Object Class

The base class that the ACE *Service Configurator* framework uses to configure & manage service implementations via the following hook methods:

- Initialize & finalize a service
- Suspend service execution temporarily & resume execution of a suspended service
- Report key service information
  - e.g., its purpose, current status, port number where it listens for client connections, etc.



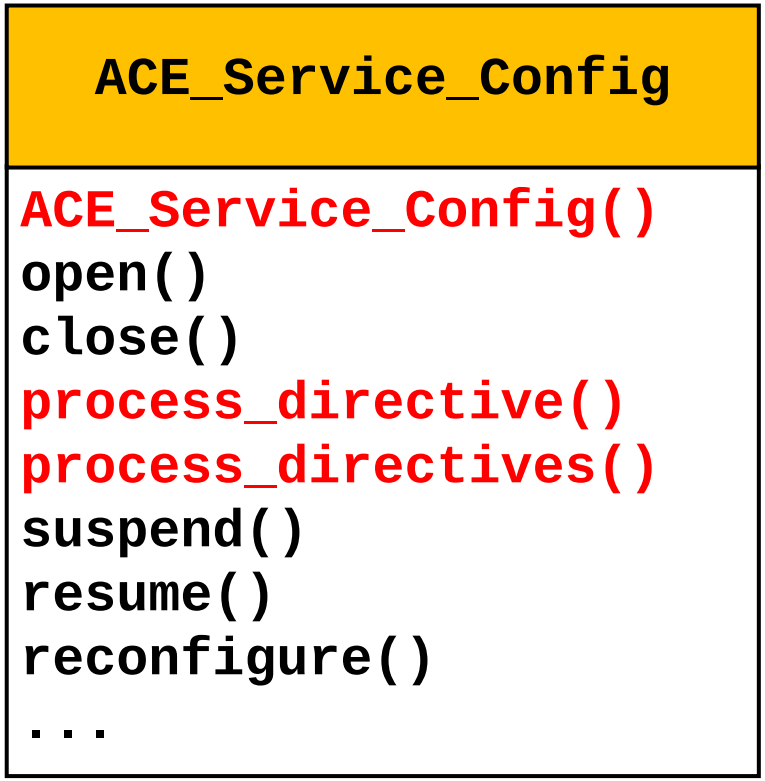
By inheriting from ***ACE_Event_Handler*** *an* ***ACE_Service_Object*** *can also be registered with ACE Reactor framework*

Handles *variability* of service-specific lifecycle behavior via a *common* API

The ACE_Service_Config Class

Implements the *Façade* pattern to integrate other *Service Configurator* classes & provide following capabilities:

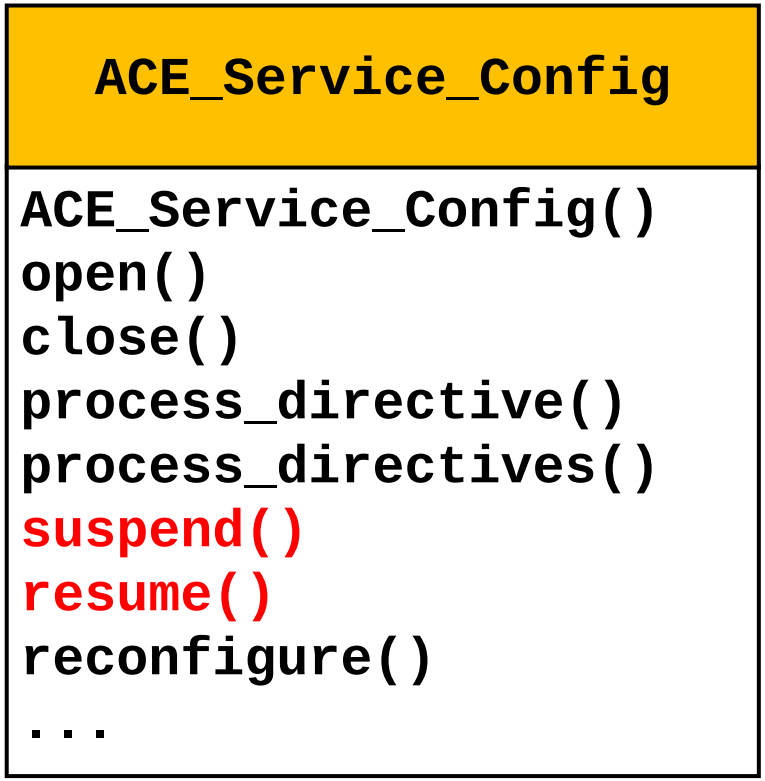
- Interprets a scripting language that provides directives to
  - Locate & initialize a service implementation at run time



The ACE_Service_Config Class

Implements the *Façade* pattern to integrate other *Service Configurator* classes & provide following capabilities:

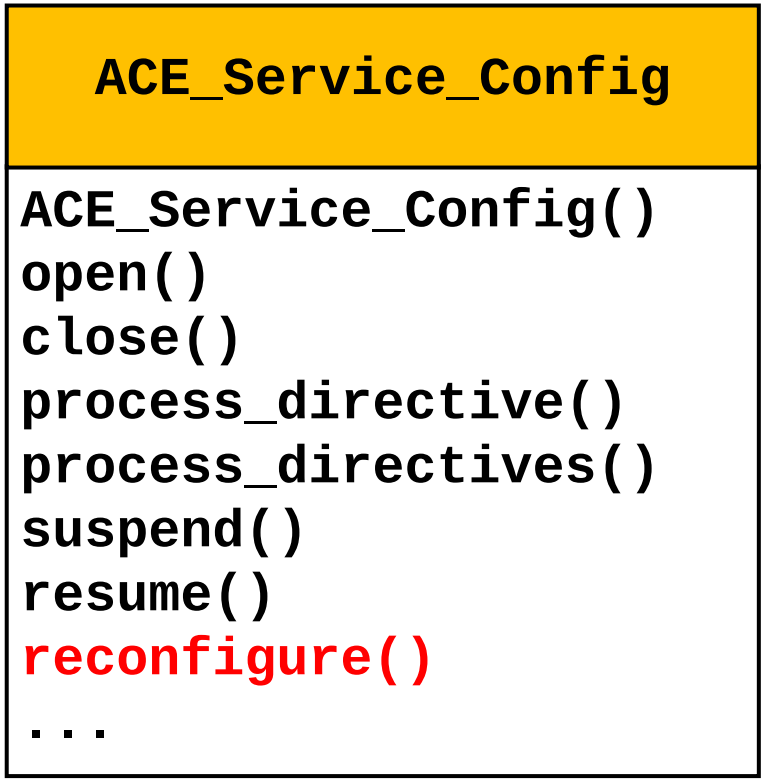
- Interprets a scripting language that provides directives to
  - Locate & initialize a service implementation at run time
  - Suspend, resume, reinitialize, & shutdown a service after it's been initialized



The ACE_Service_Config Class

Implements the *Façade* pattern to integrate other *Service Configurator* classes & provide following capabilities:

- Interprets a scripting language that provides directives to
  - Locate & initialize a service implementation at run time
  - Suspend, resume, reinitialize, & shutdown a service after it's been initialized
- It allows service reconfiguration at runtime

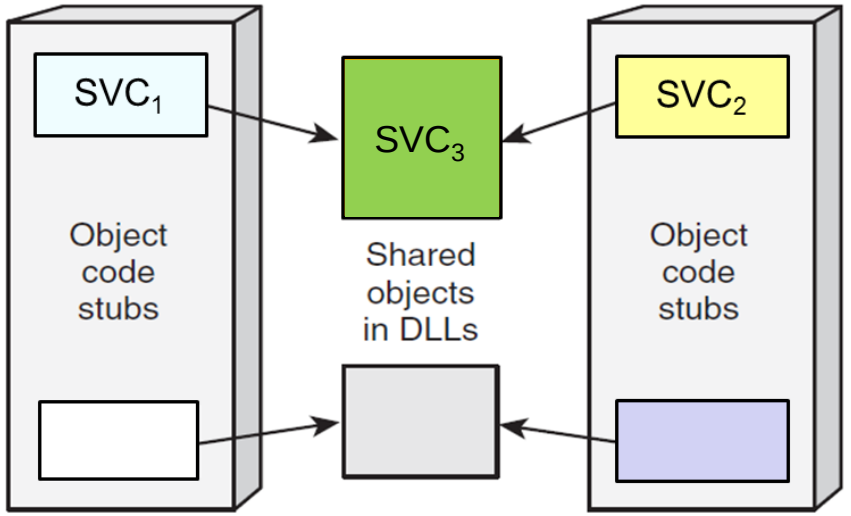


The ACE_Service_Config Class

Implements the *Façade* pattern to integrate other *Service Configurator* classes & provide following capabilities:

- Interprets a scripting language that provides directives to
  - Locate & initialize a service implementation at run time
  - Suspend, resume, reinitialize, & shutdown a service after it's been initialized
- It allows service reconfiguration at runtime
- It supports management of
  - **Static services** – Linked into an app
  - **Dynamic services** – Linked dynamically from shared libraries (DLLs)

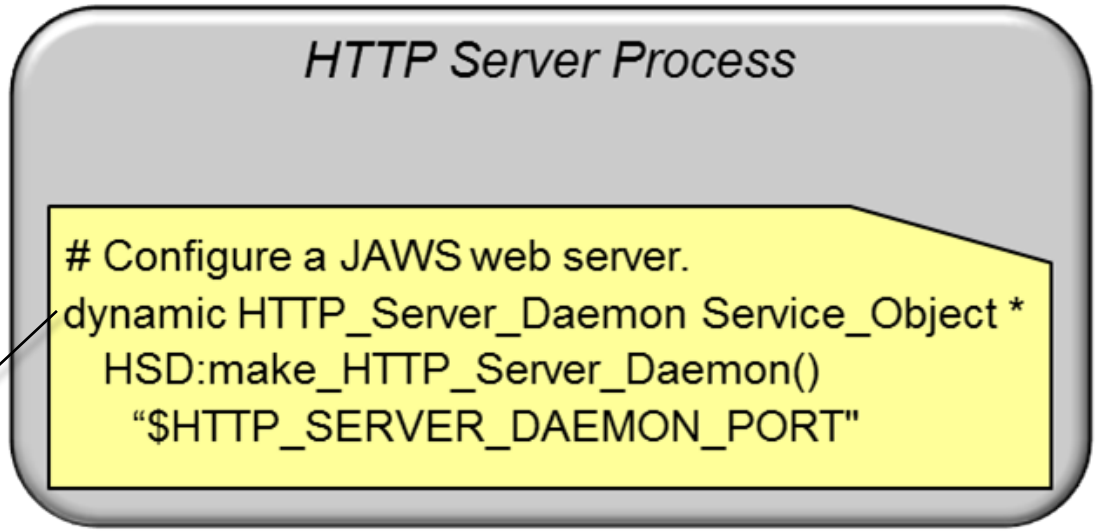
ACE_Service_Config



Handles the *variability* of static & dynamic configuration via a *common* API

Service Configuration Directives

- Directives are commands that can be passed to the ACE *Service Configurator* framework to designate its behavior



Service Configuration Directives

- Directives are commands that can be passed to the ACE *Service Configurator* framework to designate its behavior
- The following directives are supported:

Directive	Description
`dynamic`	Dynamically link a service and initialize it by calling its **`init()`** hook method
`static`	Call the **`init()`** hook method to initialize a service that was linked statically
`remove`	Remove a service completely by calling its **`fini()`** hook method & unlinking it from the app process when it's no longer used
`suspend`	Call a service's **`suspend()`** hook method to pause it without removing it
`resume`	Call a service's **`resume()`** hook method to continue processing a service that was suspended earlier
`stream`	Initialize an ordered list of hierarchically related modules

Service Configuration Directives

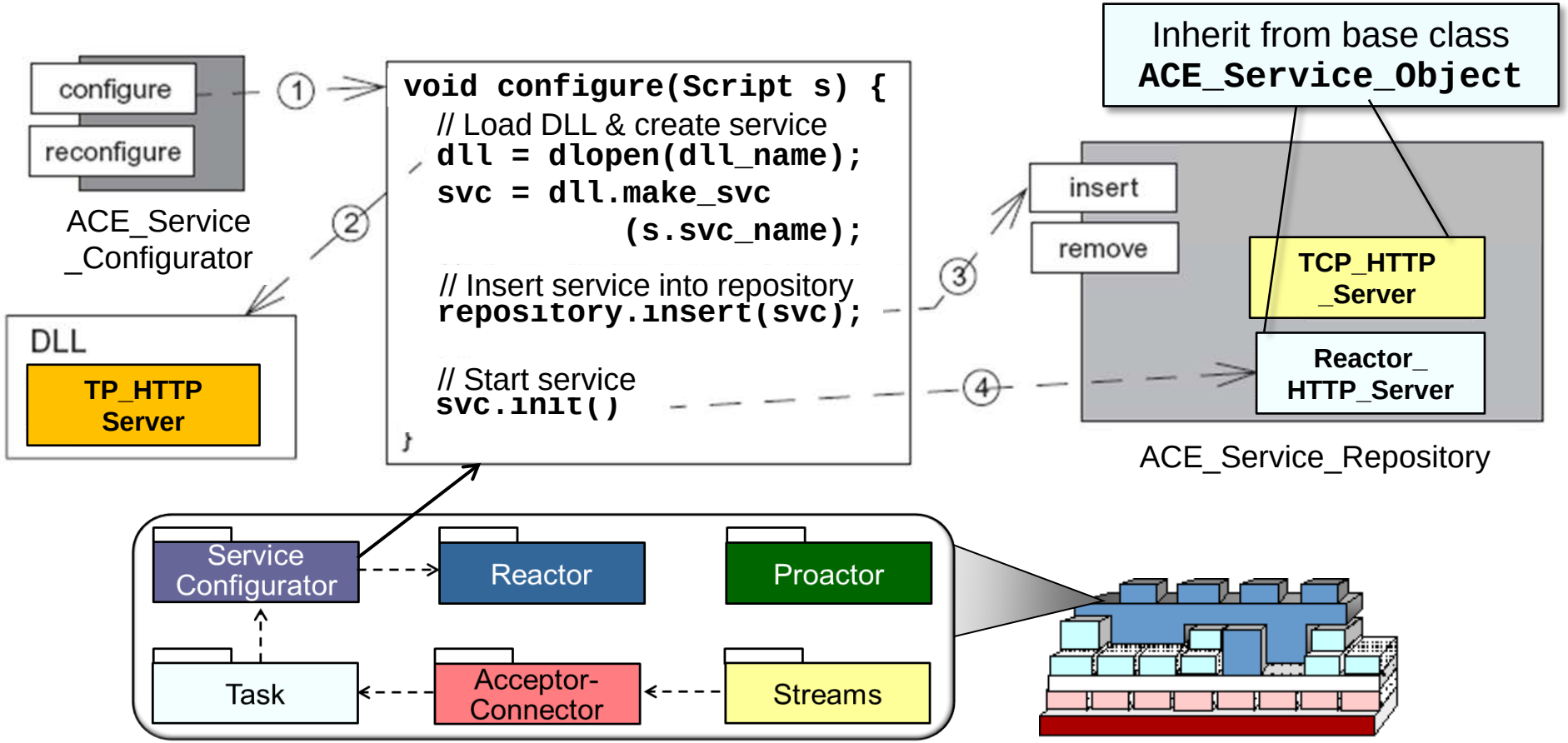
- Directives are commands that can be passed to the ACE *Service Configurator* framework to designate its behavior
- The following directives are supported:

Directive	Description
`dynamic`	Dynamically link a service and initialize it by calling its **`init()`** hook method
`static`	Call the **`init()`** hook method to initialize a service that was linked statically
`remove`	Remove a service completely by calling its **`fini()`** hook method & unlinking it from the app process when it's no longer used
`suspend`	Call a service's **`suspend()`** hook method to pause it without removing it
`resume`	Call a service's **`resume()`** hook method to continue processing a service that was suspended earlier
`stream`	Initialize an ordered list of hierarchically related modules

- Directives can be specified to **`ACE_Service_Config`** by either:
  - Using configuration files (named **`svc.conf`** by default) that contain one or more directives
  - By passing individual directives as strings to the **`ACE_Service_Config::process_directive()`** method

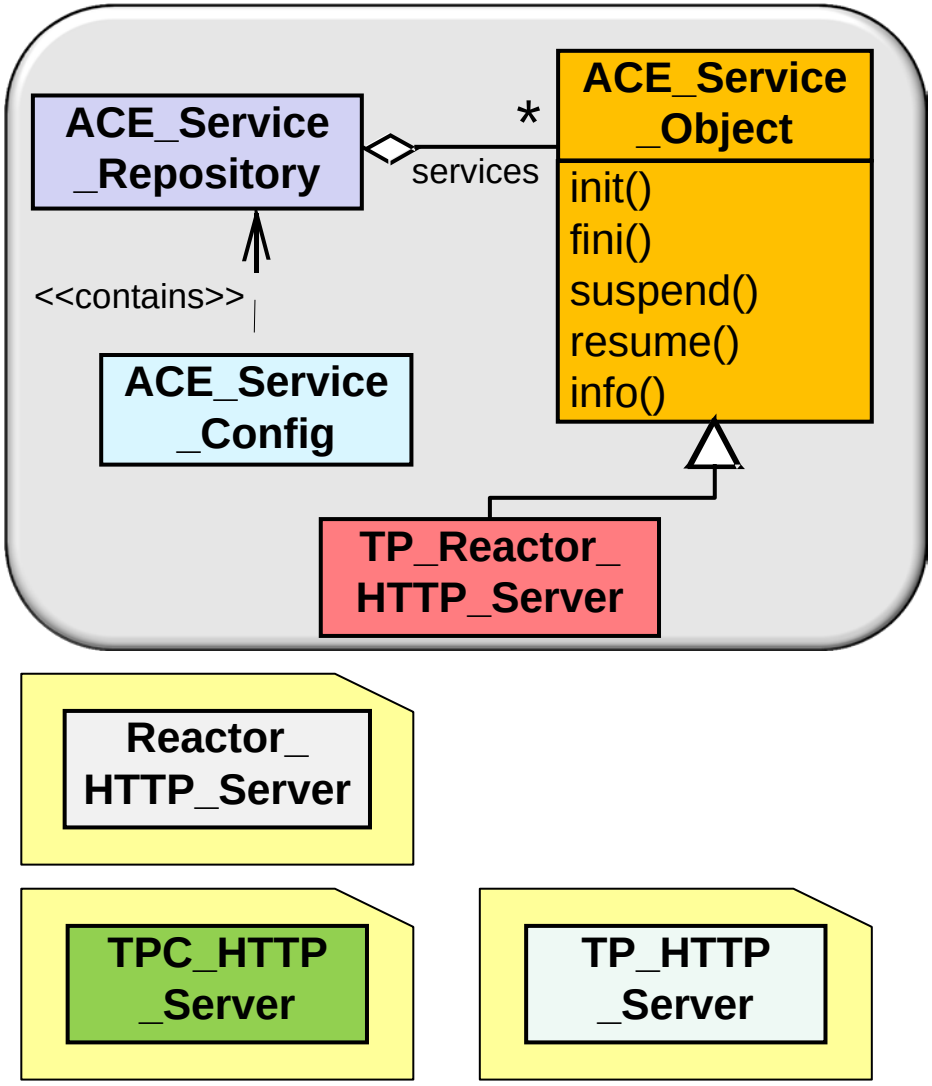
Summary

- ACE *Service Configurator* framework improves JAWS extensibility by deferring implementation choices until late in the lifecycle



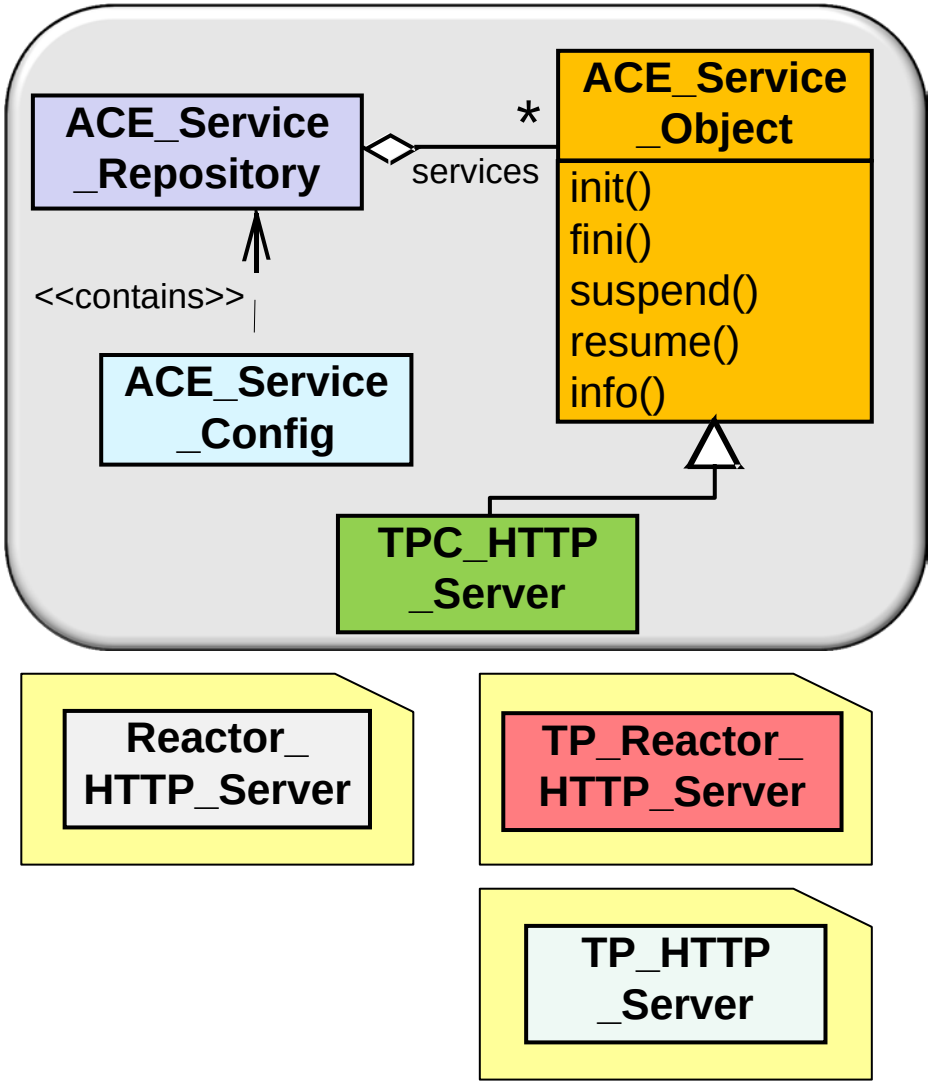
Summary

- ACE *Service Configurator* framework improves JAWS extensibility by deferring implementation choices until late in the lifecycle
- This late binding yields the following benefits:
  - Apps are composed of multiple services that can be developed independently



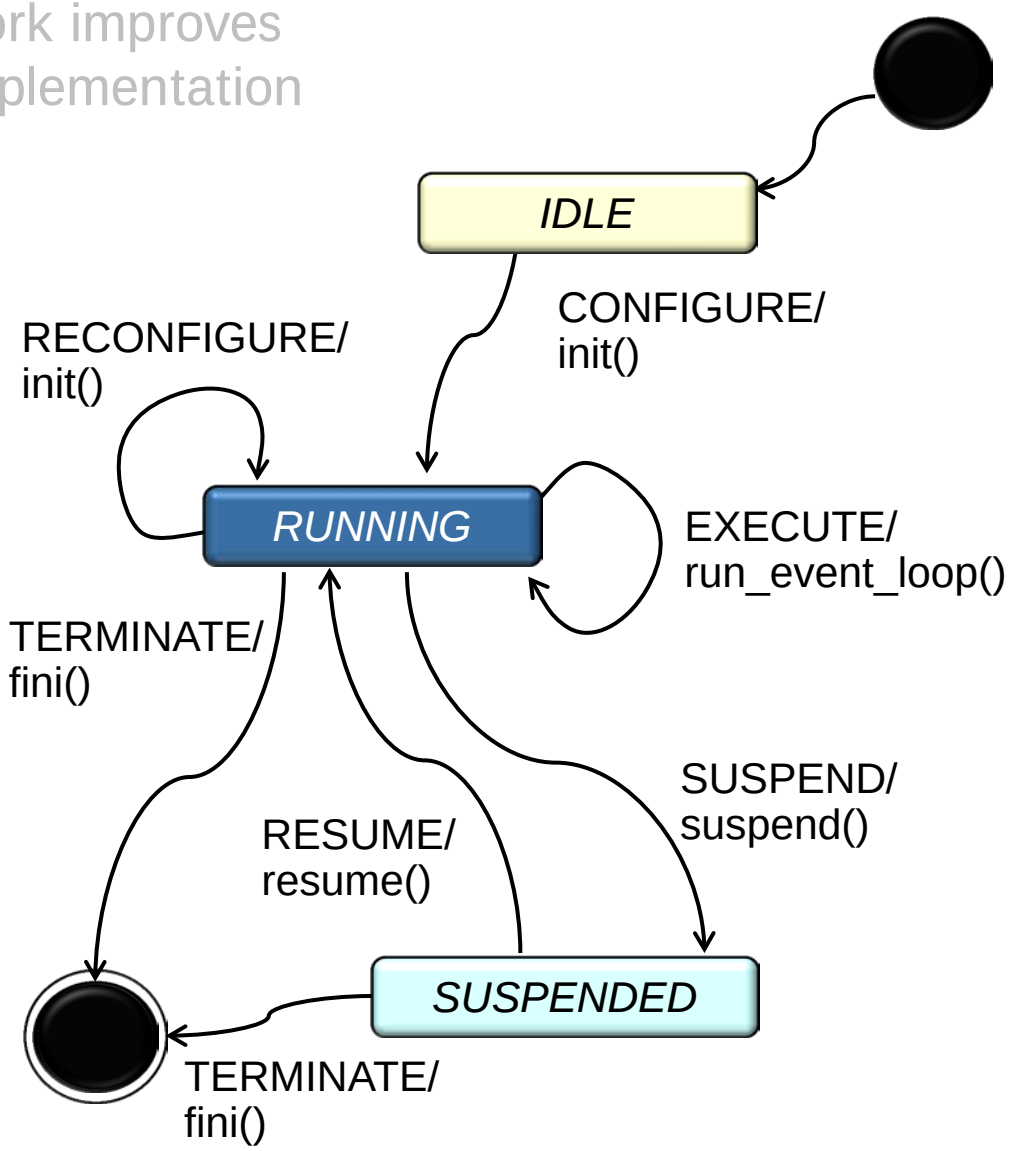
Summary

- ACE *Service Configurator* framework improves JAWS extensibility by deferring implementation choices until late in the lifecycle
- This late binding yields the following benefits:
  - Apps are composed of multiple services that can be developed independently
  - Developers can concentrate on a functionality & other key design dimensions without committing prematurely to a particular configuration



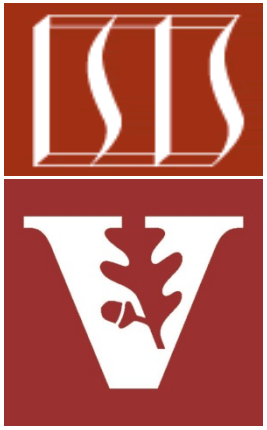
Summary

- ACE *Service Configurator* framework improves JAWS extensibility by deferring implementation choices until late in the lifecycle
- This late binding yields the following benefits:
  - Apps are composed of multiple services that can be developed independently
  - Developers can concentrate on a functionality & other key design dimensions without committing prematurely to a particular configuration
  - Apps can be (re)configured at installation-time or runtime



Patterns & Frameworks for Service Configuration & Activation: Part 3

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



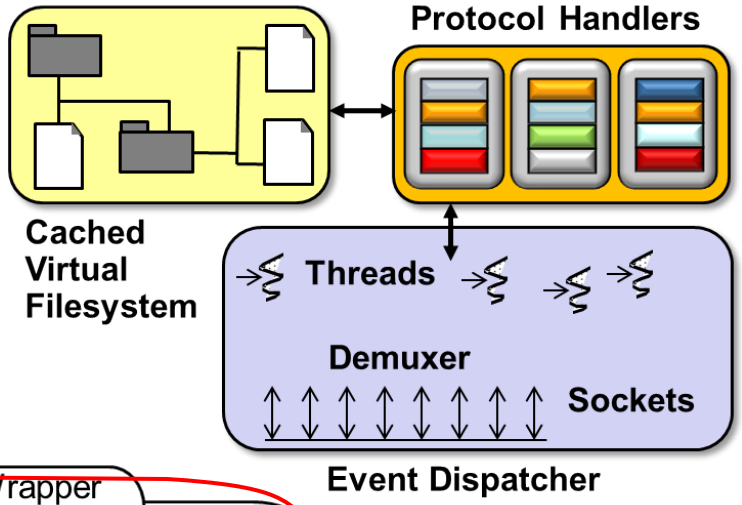
Professor of Computer Science

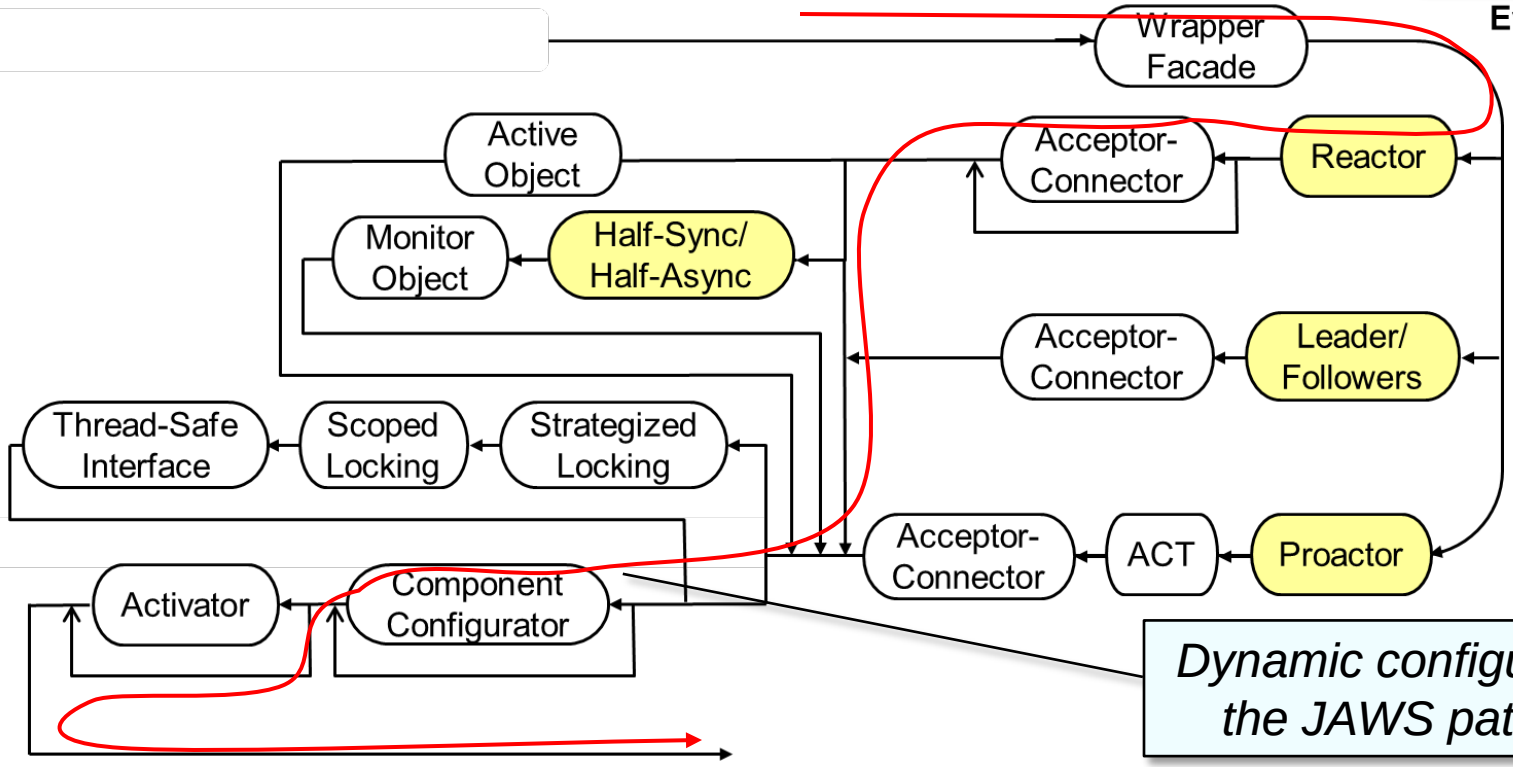
Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Topics Covered in this Part of the Module

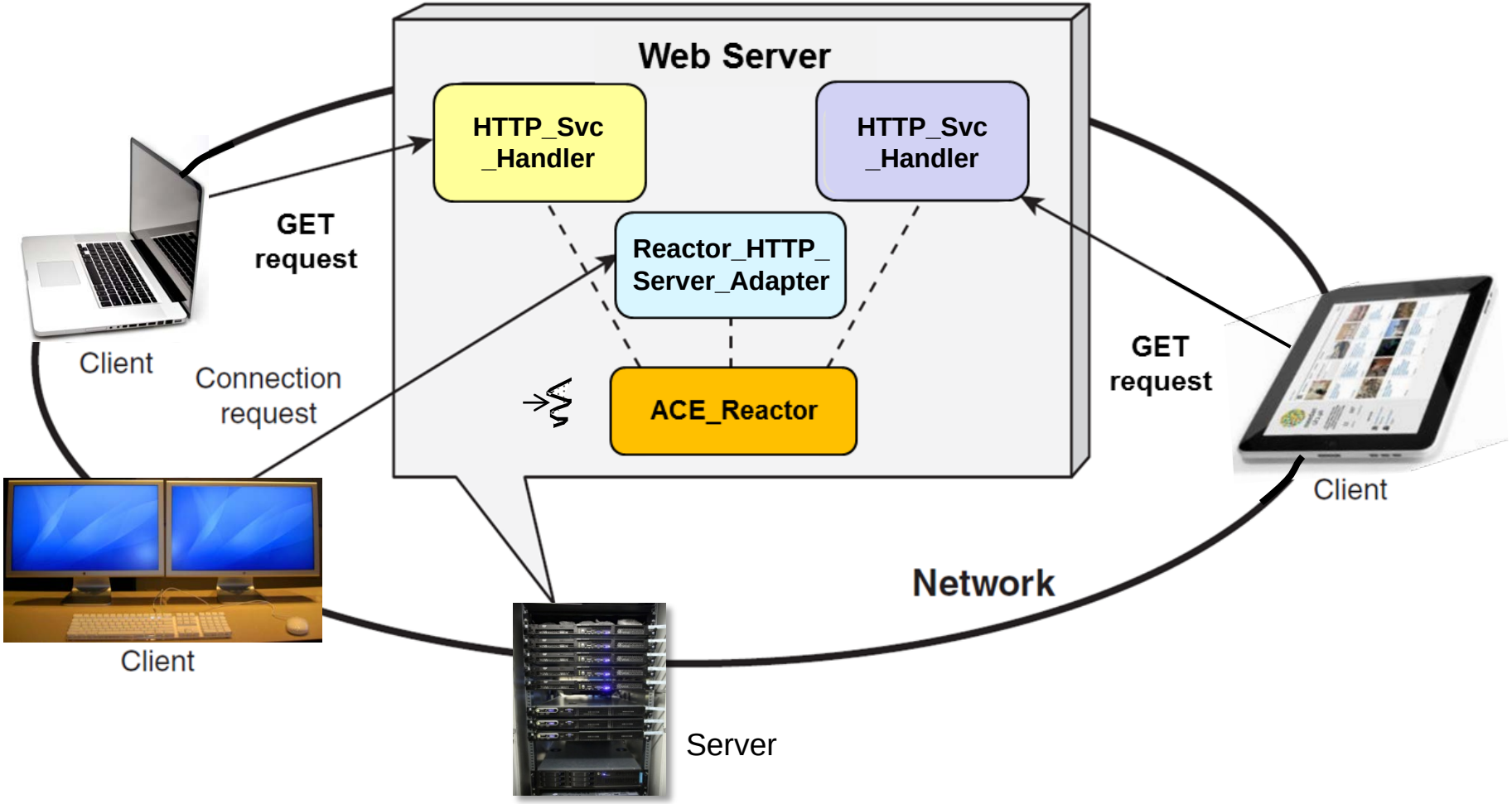
- Describe the *Component Configurator* pattern
- Describe the ACE *Service Configurator* framework
- Apply ACE Service Configurator to JAWS





Using ACE_Service_Object with JAWS

To showcase **`ACE_Service_Object`**, we'll reimplement the reactive JAWS web server from the *Acceptor-Connector* example



Components in this web server are linked dynamically at installation & runtime

Using ACE_Service_Object with JAWS

To showcase **ACE_Service_Object**, we'll reimplement the reactive JAWS web server from the *Acceptor-Connector* example

```
template <typename ACCEPTOR>
class Reactor_HTTP_Server_Adapter : public ACE_Service_Object
{
public:
  virtual int init
     (int argc, char *argv[]);
  virtual int fini ();
  virtual int info (char **, size_t) const;
  virtual int suspend ();
  virtual int resume ();

private:
  Reactor_HTTP_Server<ACCEPTOR> *server_;
};
```

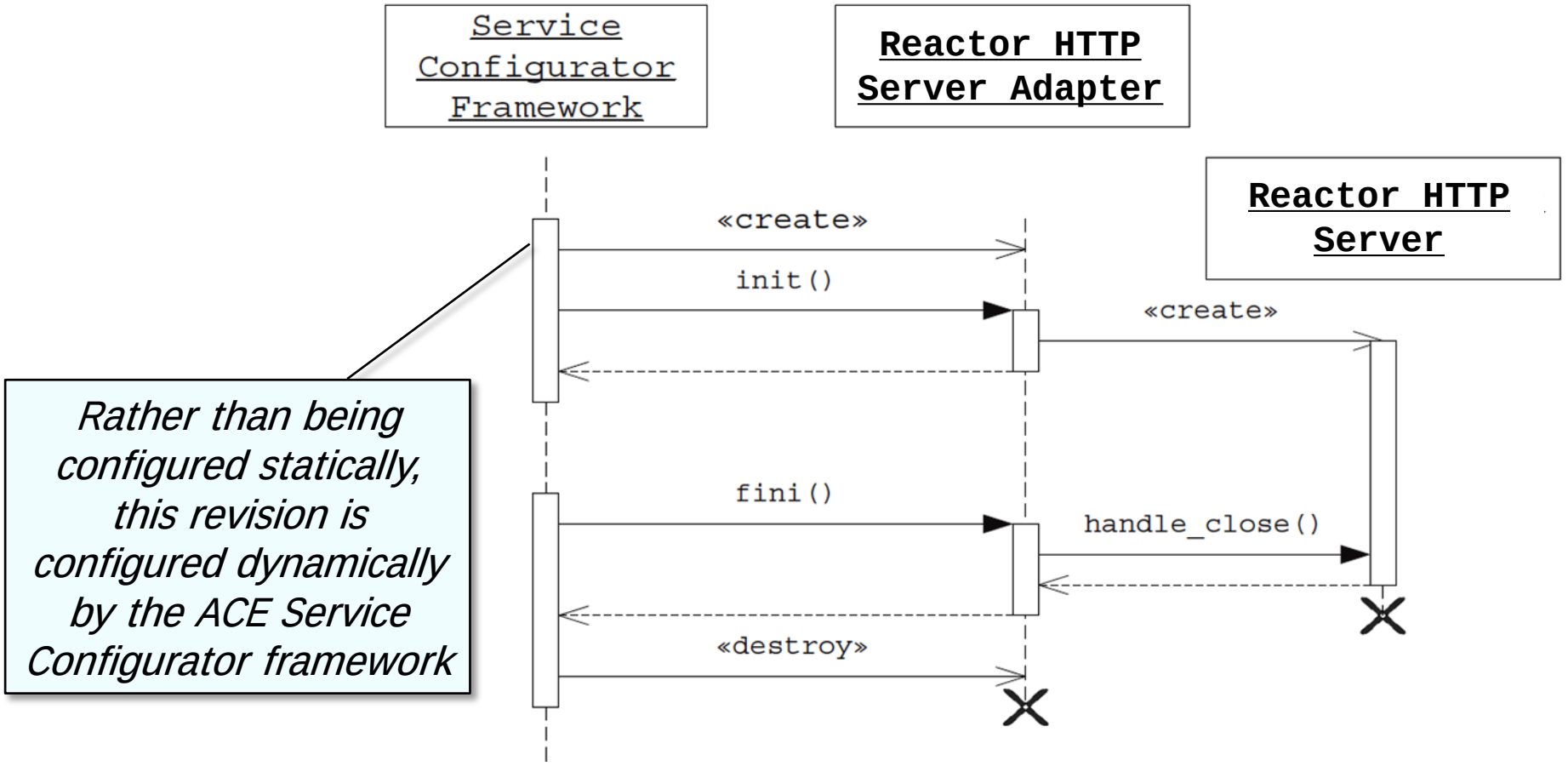
Hook methods inherited from `ACE_Service_Object`

Reuse `Reactor_HTTP_Server` from earlier examples

Note the use of the GoF *Adapter* pattern!

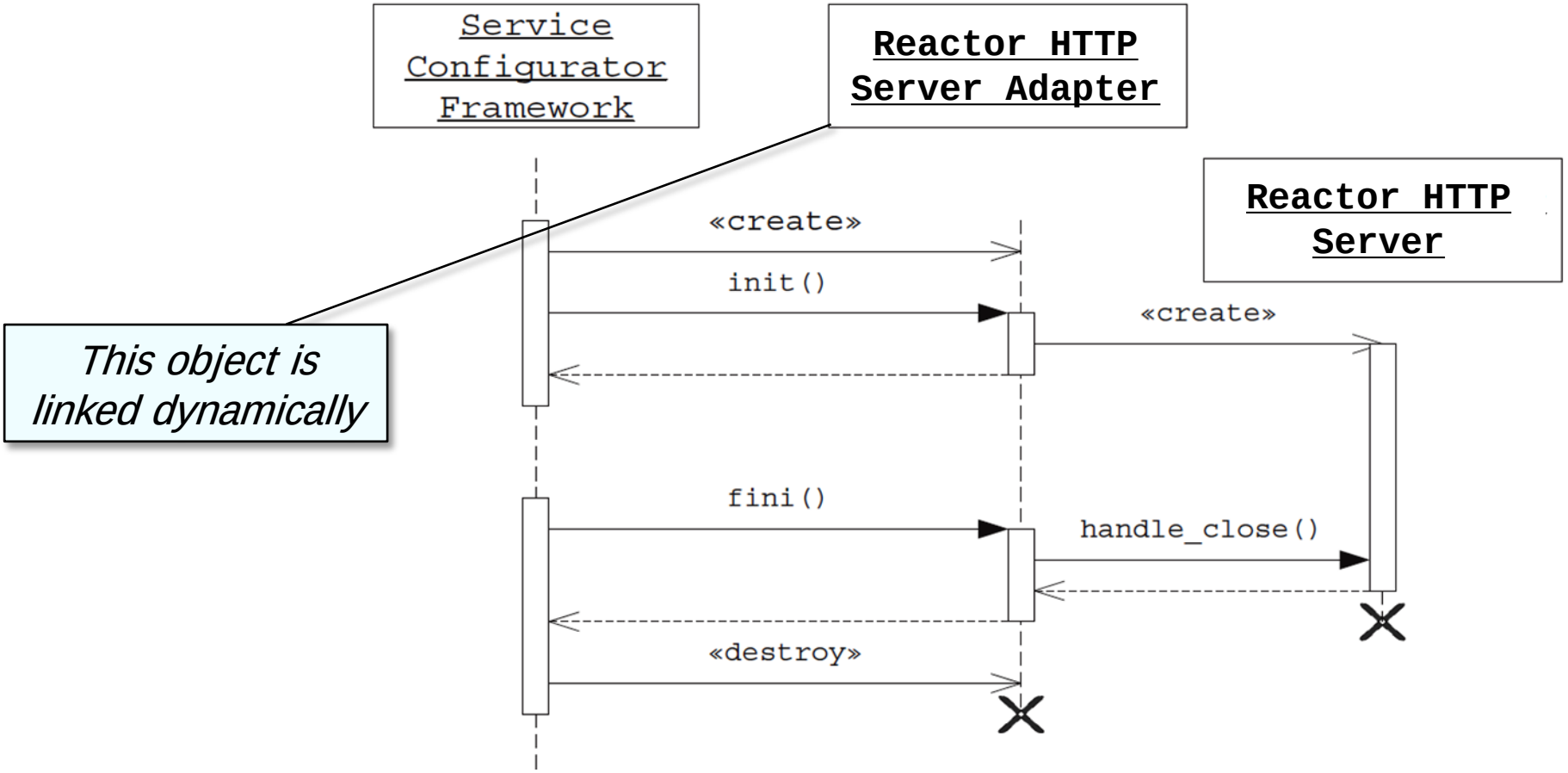
Using ACE_Service_Object with JAWS

To showcase **`ACE_Service_Object`**, we'll reimplement the reactive JAWS web server from the *Acceptor-Connector* example



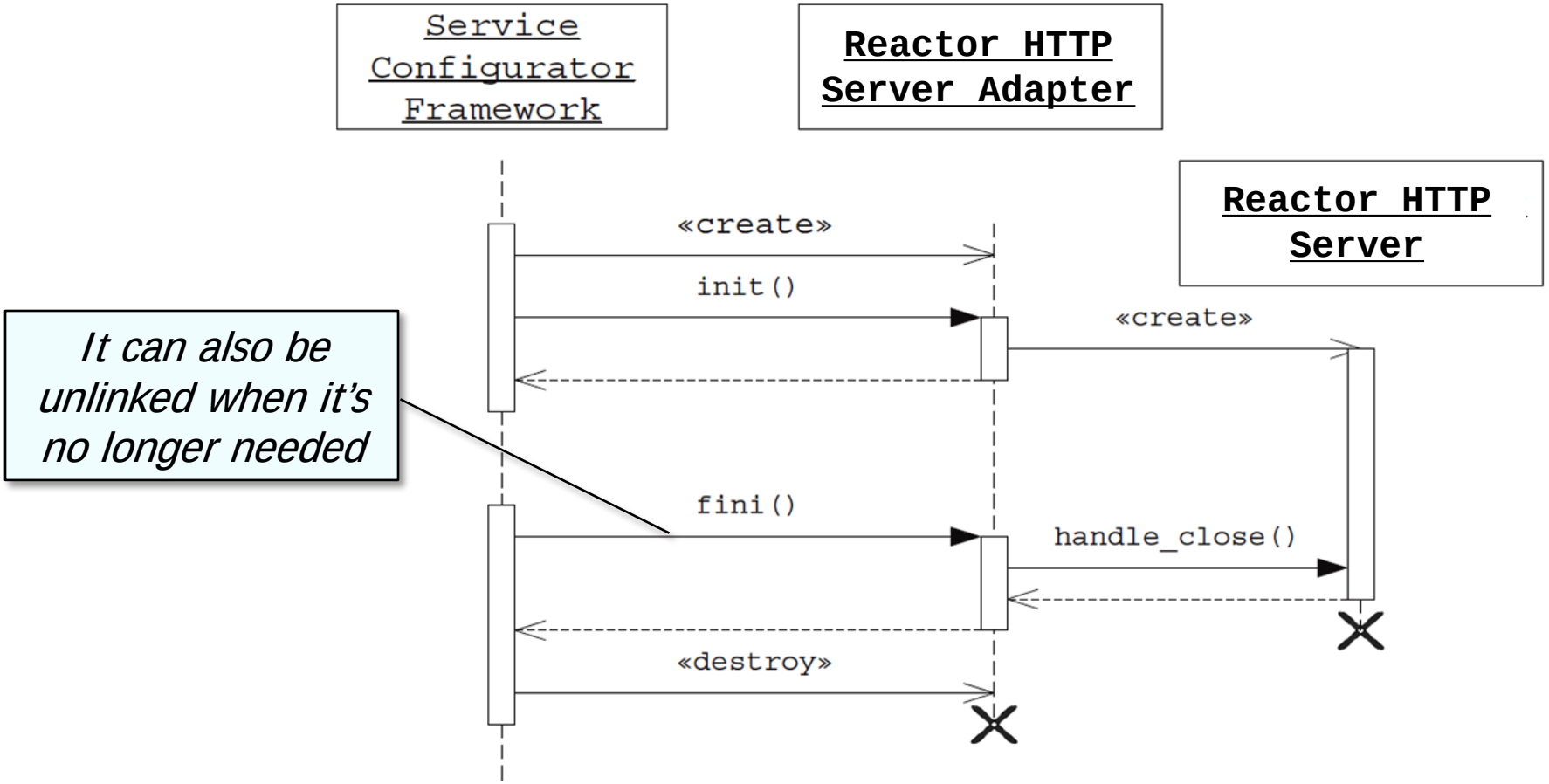
Using ACE_Service_Object with JAWS

To showcase **`ACE_Service_Object`**, we'll reimplement the reactive JAWS web server from the *Acceptor-Connector* example



Using ACE_Service_Object with JAWS

To showcase **`ACE_Service_Object`**, we'll reimplement the reactive JAWS web server from the *Acceptor-Connector* example



Using ACE_Service_Object with JAWS

Hook method called back by ACE *Service Configurator* framework to initialize the service

```
 1 template <typename ACCEPTOR> int
 2 Reactor_HTTP_Server_Adapter<ACCEPTOR>::init
 3   (int argc, char *argv[])
 4 {
 5   server_ = new (nothrow)
 6               Reactor_HTTP_Server<ACCEPTOR>
 7               (argc, argv,
 8                ACE_Reactor::instance ()));

 9   return server_ == 0 ? -1 : 0;
10 }
```

Allocate & initialize the `Reactor_HTTP_Server`

Returning -1 causes this service object to be destroyed

Using ACE_Service_Object with JAWS

```
template <typename ACCEPTOR> int
Reactor_HTTP_Server_Adapter<ACCEPTOR>::fini ()
{
  server_->handle_close (); server_ = 0; return 0;
}
```

This hook method is called by framework to terminate the service

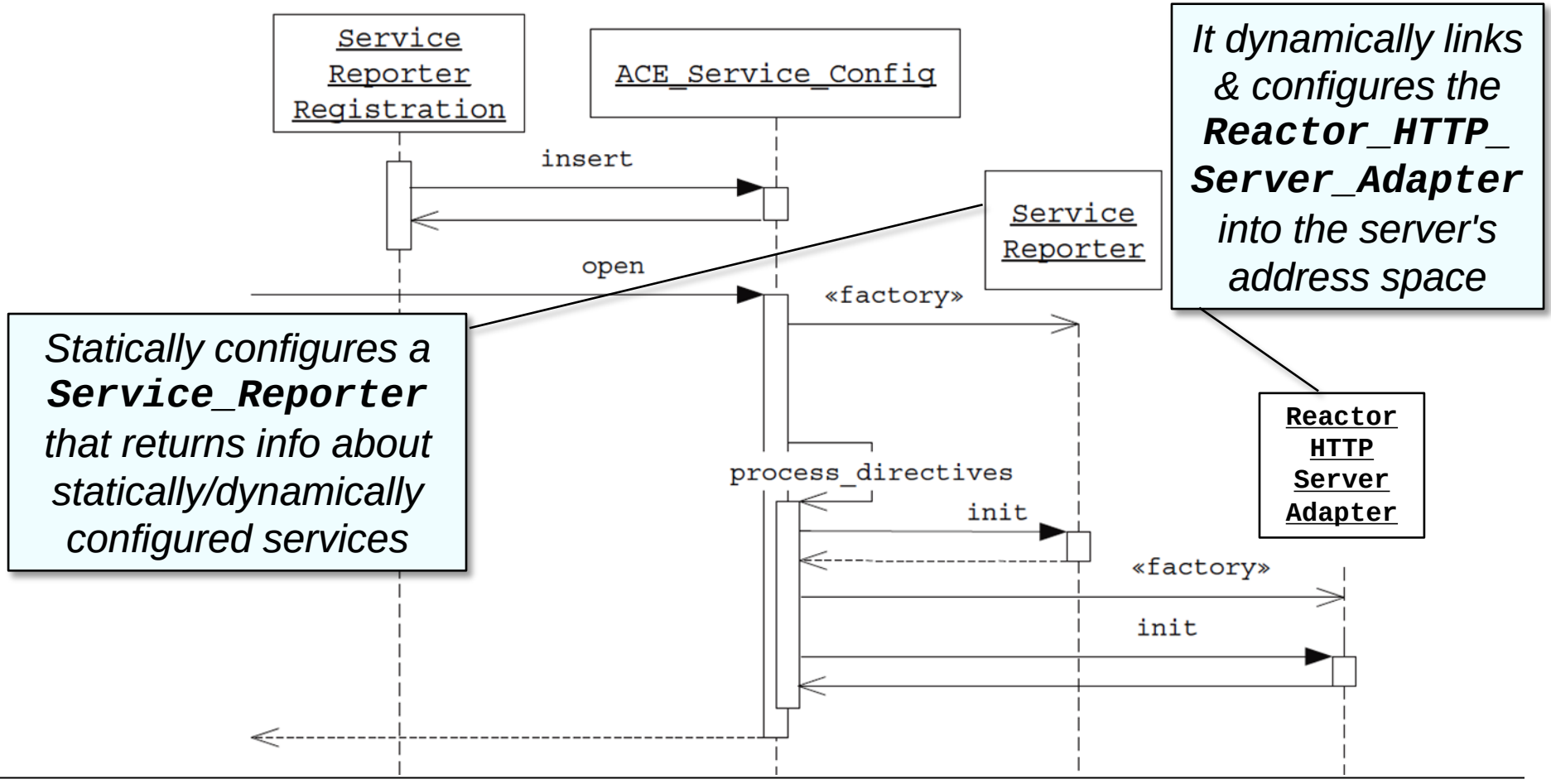
```
template <typename ACCEPTOR> int
Reactor_HTTP_Server_Adapter<ACCEPTOR>::suspend ()
{
   return server_->reactor ()->suspend_handler (server_);
}
```

These hook methods are called by framework to suspend/resume a service

```
template <typename ACCEPTOR> int
Reactor_HTTP_Server_Adapter<ACCEPTOR>::resume ()
{
  return server_->reactor ()->resume_handler (server_);
}
```

Using ACE_Service_Config with JAWS

Apply the ACE *Service Configurator* framework to create a server whose initial configuration behaves as follows:



Shortly we'll show how to dynamically reconfigure another reactive web service

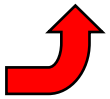
Using ACE_Service_Config with JAWS

Here's a generic **`main()`** program that is useful for many apps & services

```
 1 #include "ace/OS.h"
 2 #include "ace/Service_Config.h"
 3 #include "ace/Reactor.h"

 4 int main (int argc, char *argv[]) {
 5   ACE_STATIC_SVC_REGISTER (Reporter);

 6   ACE_Service_Config::open
 7     (argc, argv, ACE_DEFAULT_LOGGER_KEY, 0);
 8
 9   ACE_Reactor::instance ()->run_reactor_event_loop ();
10   return 0;
11 }
```

Uses `svc.conf` file to configure the `Service_Reporter` & `Reactor_HTTP_Server_Adapter` into app process

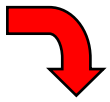
Run the reactor's event loop



Using ACE_Service_Config with JAWS

```
#include "Reactor_HTTP_Server_Adapter.h"
#include "HTTP_Svc_Acceptor.h"
#include "HTTP_Server_Daemon_export.h"
```

Instantiate the adapter using the `HTTP_Svc_Acceptor` from our previous example



```
typedef Reactor_HTTP_Server_Adapter<HTTP_Svc_Acceptor>
        HTTP_Server_Daemon;

ACE_FACTORY_DEFINE (HSD, HTTP_Server_Daemon)
```

This macro ensures the `HTTP_Server_Daemon` type can be linked dynamically

Using ACE_Service_Config with JAWS

```
#include "Reactor_HTTP_Server_Adapter.h"
#include "HTTP_Svc_Acceptor.h"
#include "HTTP_Server_Daemon_export.h"



typedef Reactor_HTTP_Server_Adapter<HTTP_Svc_Acceptor>
        HTTP_Server_Daemon;

ACE_FACTORY_DEFINE (HSD, HTTP_Server_Daemon)
```

This svc.conf file is used to configure the main program

```
1 static Service_Reporter "-p $SERVICE_REPORTER_PORT"
```

Statically configure `Service_Reporter`

Dynamically configure the web server

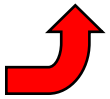
```
2 dynamic HTTP_Server_Daemon Service_Object *
3 HSD:_make_HTTP_Server_Daemon()
4   "$HTTP_SERVER_DAEMON_PORT"
```

Using ACE_Service_Config with JAWS

```
#include "Reactor_HTTP_Server_Adapter.h"
#include "HTTP_Svc_Acceptor.h"
#include "HTTP_Server_Daemon_export.h"



typedef Reactor_HTTP_Server_Adapter<HTTP_Svc_Acceptor>
        HTTP_Server_Daemon;

ACE_FACTORY_DEFINE (HSD, HTTP_Server_Daemon)
```

```
1 static Service_Reporter "-p $SERVICE_REPORTER_PORT"
```

The `ACE_Service_Config` interpreter expands these environment variables



```
2 dynamic HTTP_Server_Daemon Service_Object *
3 HSD:_make_HTTP_Server_Daemon()
4   "$HTTP_SERVER_DAEMON_PORT"
```

Using ACE Service Configurator Reconfiguration

- The previous JAWS web server configuration has the limitation that the **`ACE_Reactor::run_reactor_event_loop()`** can't be shut down on the reactor singleton

```
 1 #include "ace/OS.h"
 2 #include "ace/Service_Config.h"
 3 #include "ace/Reactor.h"

 4 int main (int argc, char *argv[]) {
 5   ACE_STATIC_SVC_REGISTER (Reporter);

 6   ACE_Service_Config::open
 7     (argc, argv, ACE_DEFAULT_LOGGER_KEY, 0);

 8   ACE_Reactor::instance ()->run_reactor_event_loop ();
 9   return 0;
10 }
```

Doesn't terminate gracefully!

Using ACE Service Configurator Reconfiguration

- The previous JAWS web server configuration has the following limitation:
  - The `ACE_Reactor::run_reactor_event_loop()` can't be shut down on the reactor singleton
- We can add these capabilities without affecting existing code or the **`Service_Reporter`** service by defining a new **`svc.conf`** file & instructing the server to reconfigure itself

Shutdown the existing web server

```
1 remove HTTP_Server_Daemon

2 dynamic HTTP_Server_Daemon Service_Object *
3 HSDex:_make_HTTP_Server_Daemon()
4   "$SERVER_SERVER_DAEMON_PORT"

5 dynamic Server_Shutdown Service_Object *
6 HSDex:_make_Server_Shutdown()
```

Restart the web server

Dynamically link new shutdown service

Using ACE Service Configurator Reconfiguration

```
class Server_Shutdown : public ACE_Service_Object {
public:
 virtual int init (int, char *[]) {
    reactor_ = ACE_Reactor::instance ();
    return ACE_Thread_Manager::instance ()->spawn (controller,
                                                   reactor_);
  }

 virtual int fini () {
    Quit_Handler *quit_handler = new Quit_Handler (reactor_);
    return reactor_->notify (quit_handler);
  }

  // ... Other method omitted ...
private:
  ACE_Reactor *reactor_;
};

ACE_FACTORY_DEFINE (HSDex, Server_Shutdown)
```

Spawn a thread to run the reactor

Notify the reactor to shut itself down

This macro ensures the `Server_Shutdown` type can be linked dynamically

Using ACE Service Configurator Reconfiguration

Runs in a separate thread of control

```
1 static void *controller (void *arg) {
2   ACE_Reactor *reactor = static_cast<ACE_Reactor *> (arg);
3   Quit_Handler *quit_handler = new Quit_Handler (reactor);

4   for (;;) {
5     std::string user_input;
6     std::getline (cin, user_input, '\n');
7     if (user_input == "quit") {
8       reactor->notify (quit_handler);
9       break;
10    }
11  }
12  return 0;
13 }
```

Use the Reactor's notify pipe to wakeup the reactor & inform it to shut down by calling `handle_exception()`

Using ACE Service Configurator Reconfiguration

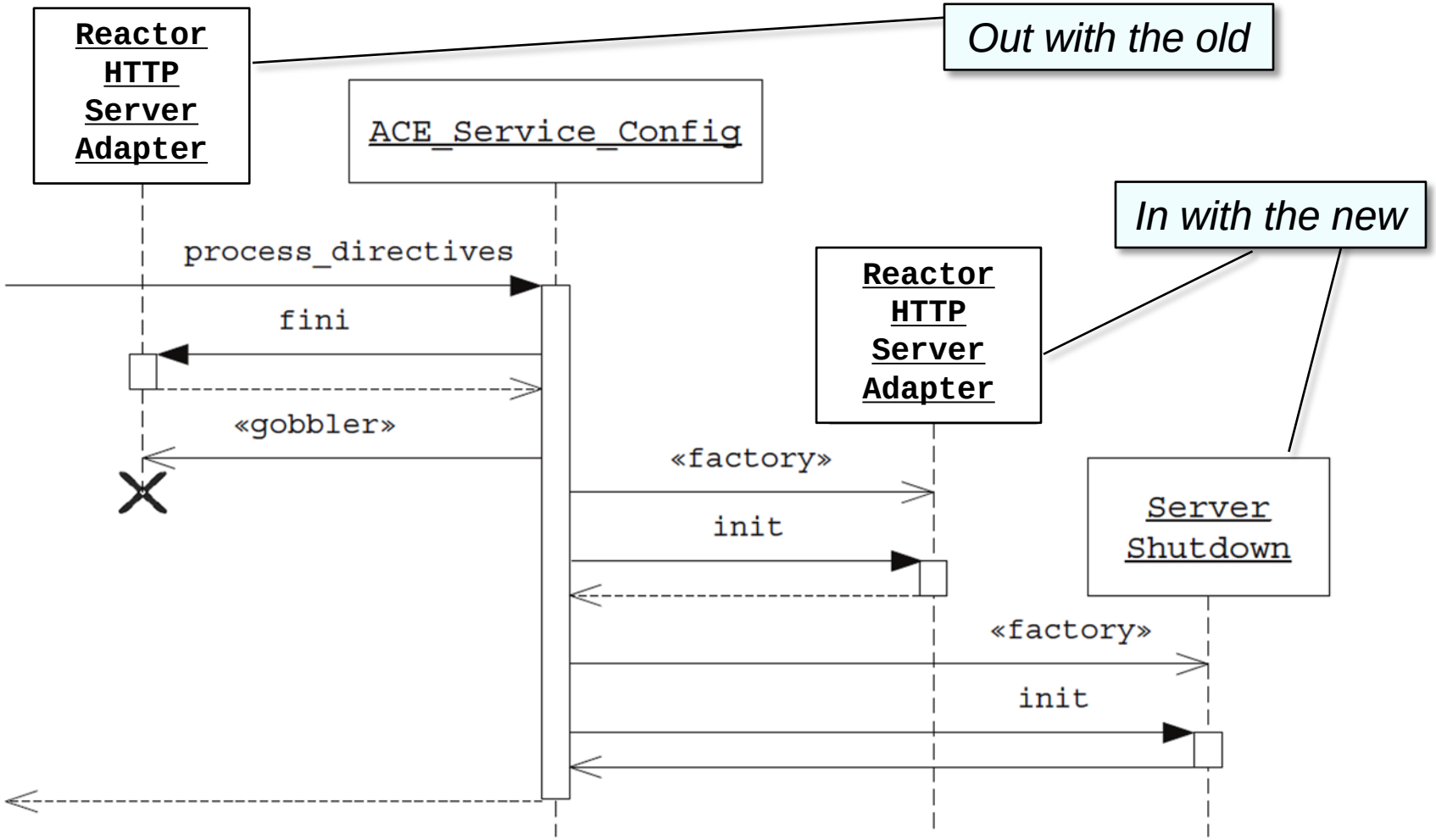
```
class Quit_Handler : public ACE_Event_Handler {
public:
  Quit_Handler (ACE_Reactor *r): ACE_Event_Handler (r) {}

  virtual int handle_exception (ACE_HANDLE) {
    reactor ()->end_reactor_event_loop ();
    return -1;
  }

  virtual int handle_close (ACE_HANDLE, ACE_Reactor_Mask) {
    delete this;
    return 0;
  }

private:
  virtual ~Quit_Handler () {}
};
```

Shutdown the reactor event loop & trigger call to `handle_close()` method

It's ok to "delete this" in this context

Private destructor ensures dynamic allocation

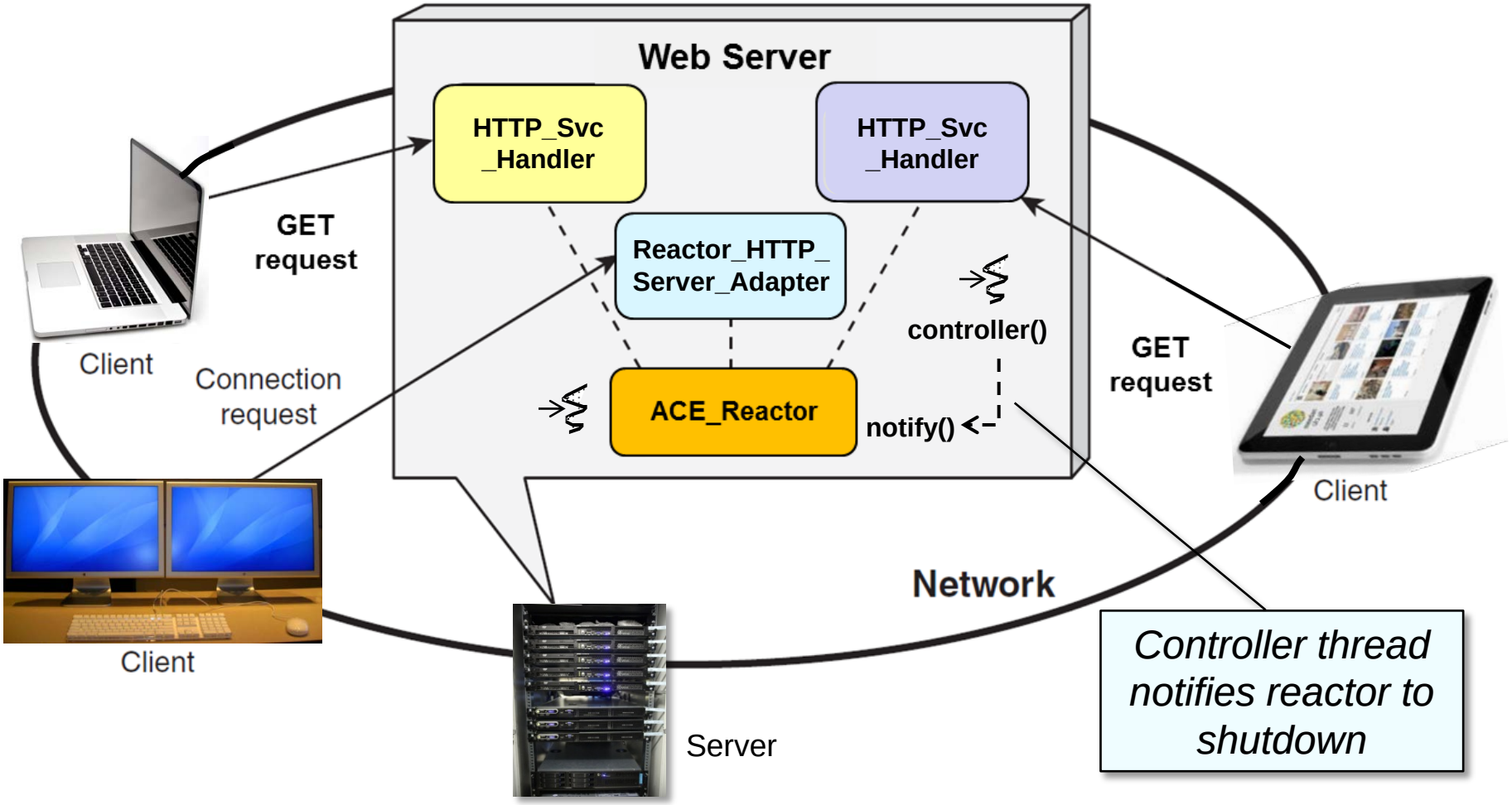
Using ACE Service Configurator Reconfiguration



Sequence diagram for ACE *Service Configurator* reconfiguration

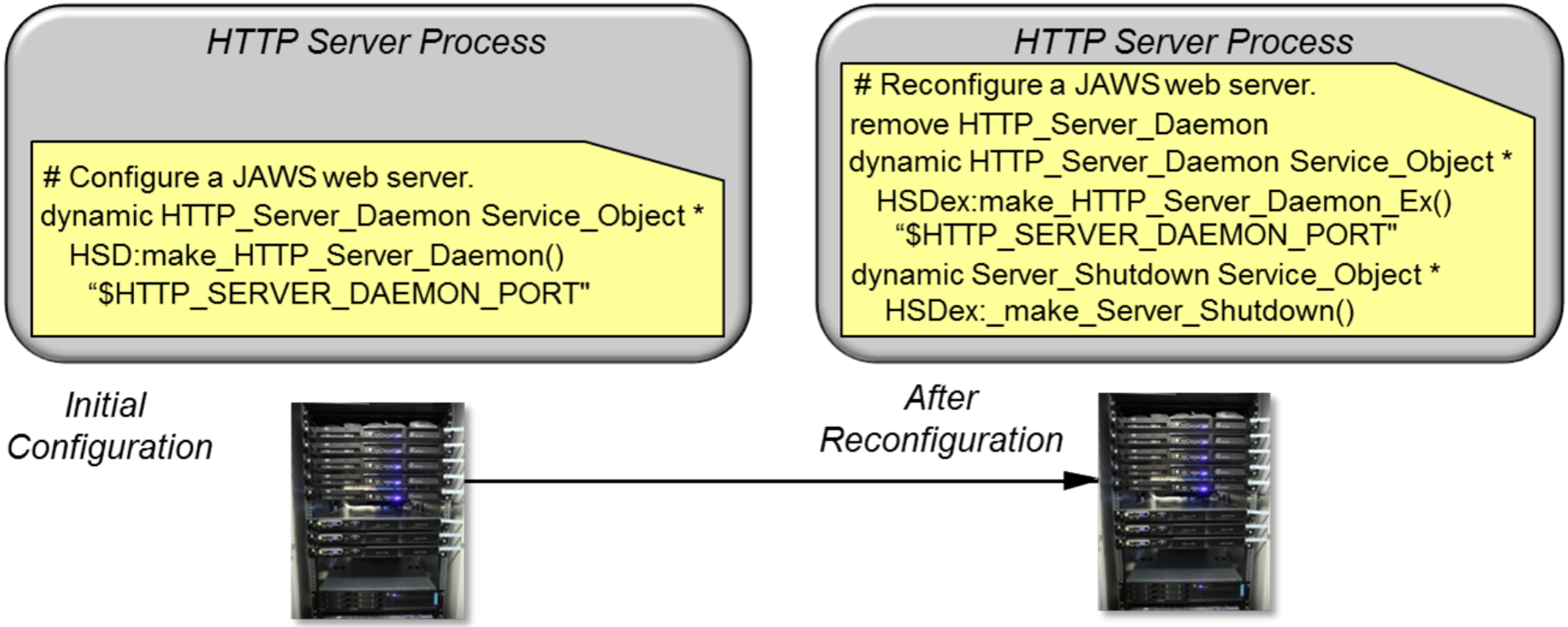
Using ACE Service Configurator Reconfiguration

The reconfigured server has a separate thread to control shutdowns gracefully



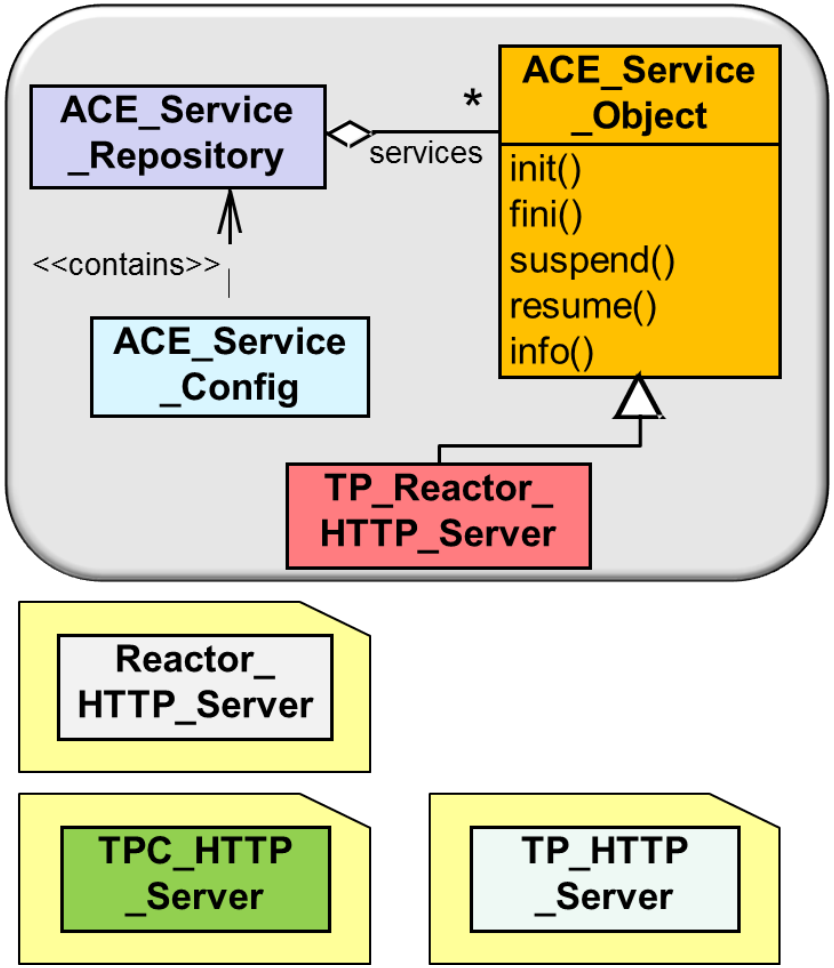
Summary

- The ACE *Service Configurator* can support the (re)configuration new services & new service implementations during installation or even at runtime



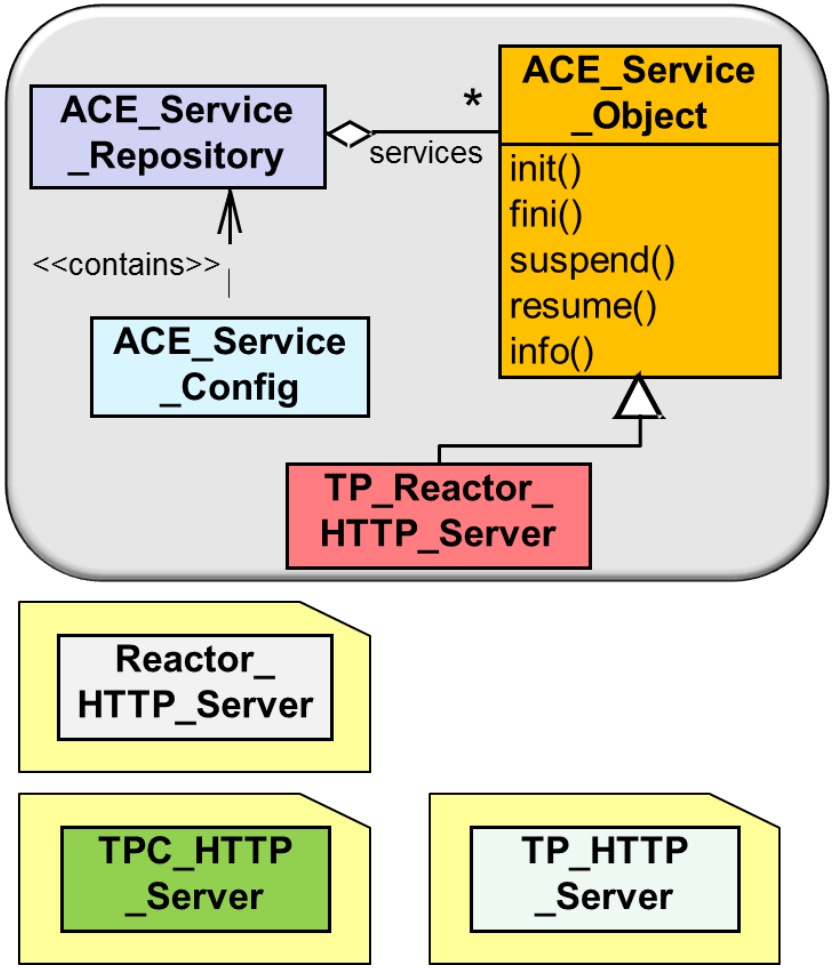
Summary

- The ACE *Service Configurator* can support the (re)configuration new services & new service implementations during installation or even at runtime
- We used these capabilities to separate parts of previous web servers implementations into independently linkable & configurable services



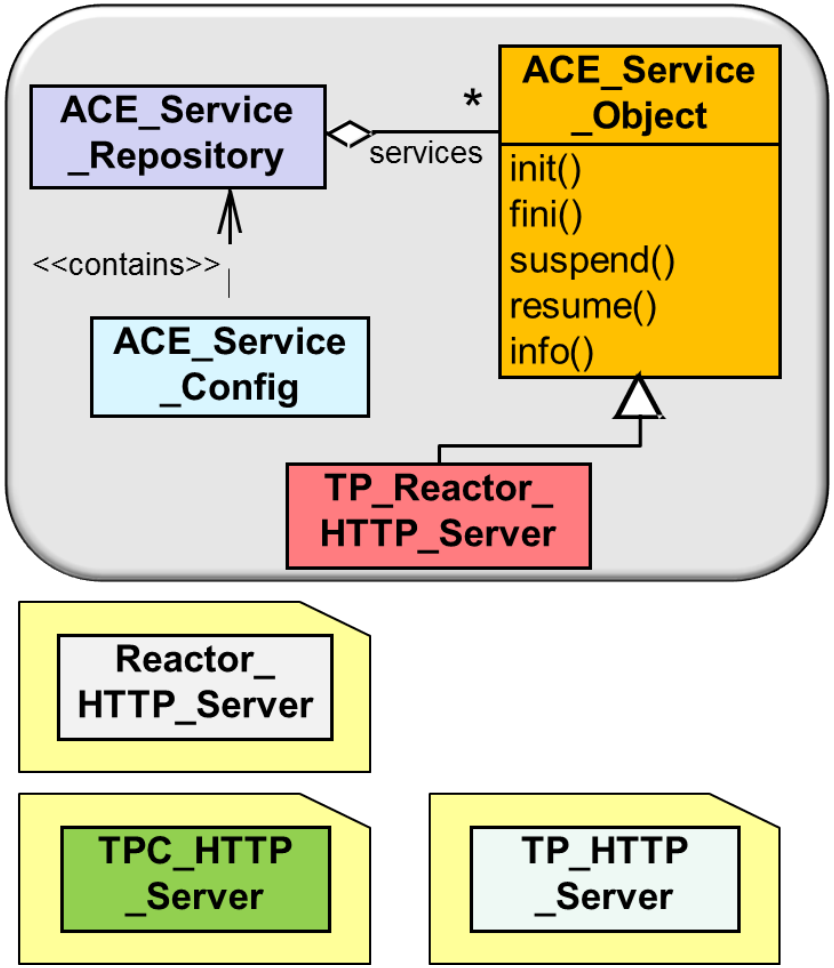
Summary

- The ACE *Service Configurator* can support the (re)configuration new services & new service implementations during installation or even at runtime
- We used these capabilities to separate parts of previous web servers implementations into independently linkable & configurable services
- The result was a web server framework that can be configured & deployed in various ways



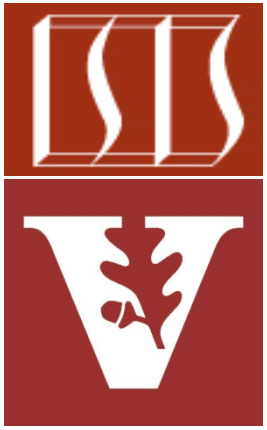
Summary

- The ACE *Service Configurator* can support the (re)configuration new services & new service implementations during installation or even at runtime
- We used these capabilities to separate parts of previous web servers implementations into independently linkable & configurable services
- The result was a web server framework that can be configured & deployed in various ways
- The ACE *Service Configurator* framework allows administrators to select features & alternative strategies that make the most sense in a particular context



Patterns & Frameworks for Service Configuration & Activation: Part 4

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



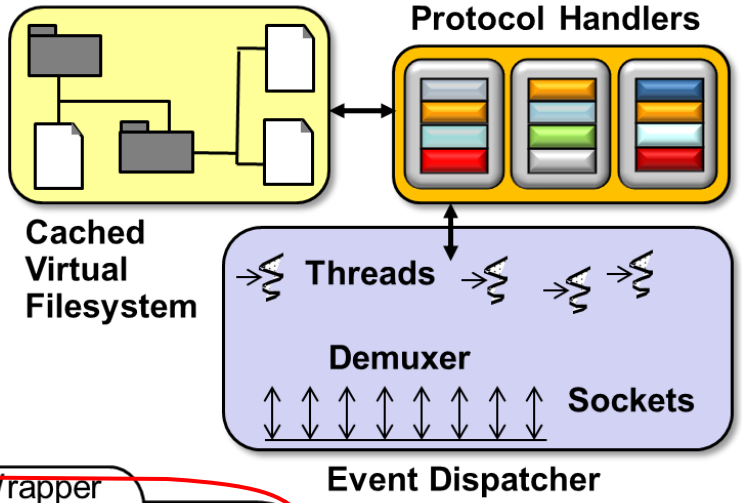
Professor of Computer Science

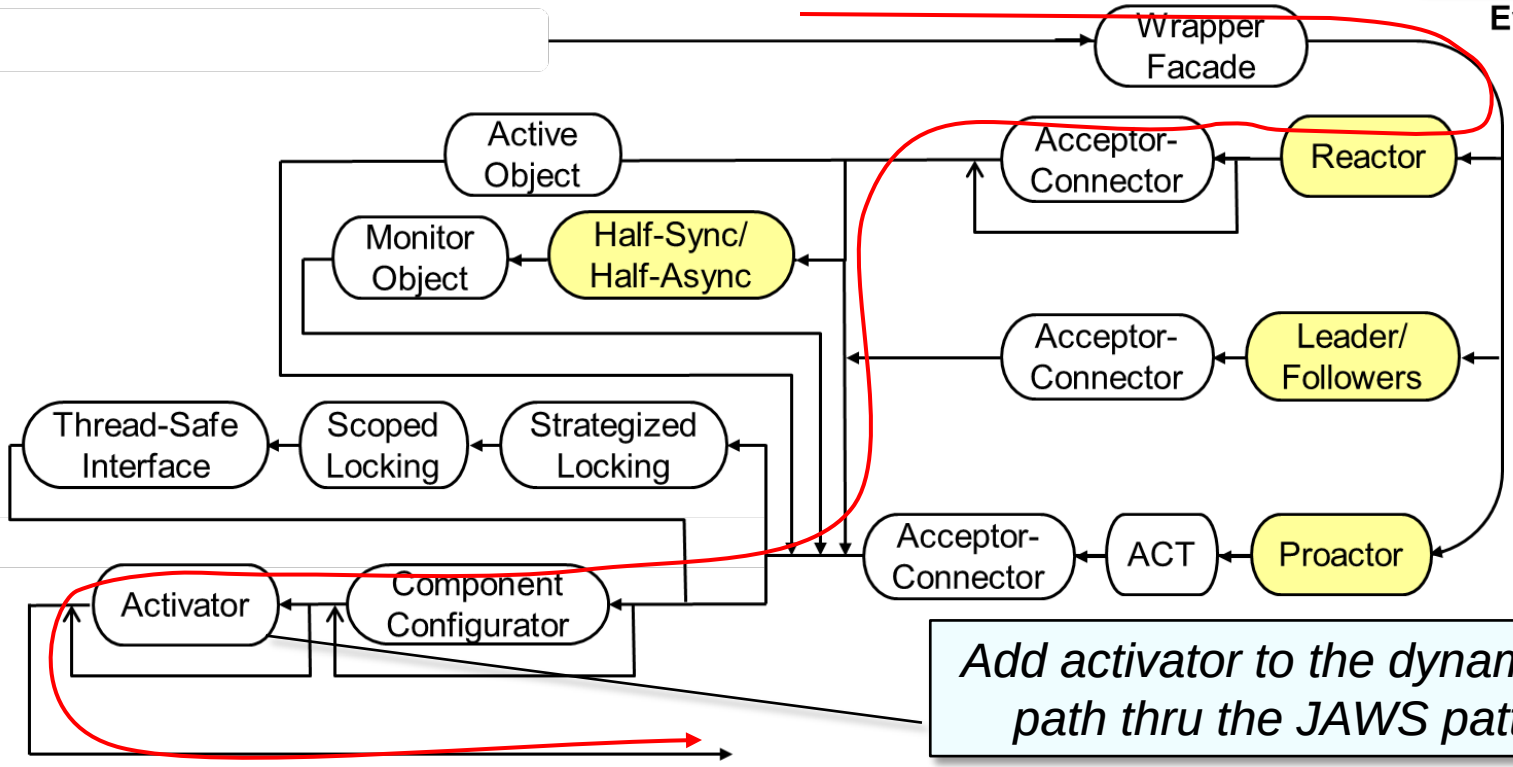
Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Topics Covered in this Part of the Module

- Describe the *Component Configurator* pattern
- Describe the ACE *Service Configurator* framework
- Apply ACE Service Configurator to JAWS
- Apply the *Activator* pattern to JAWS via the Inetd framework

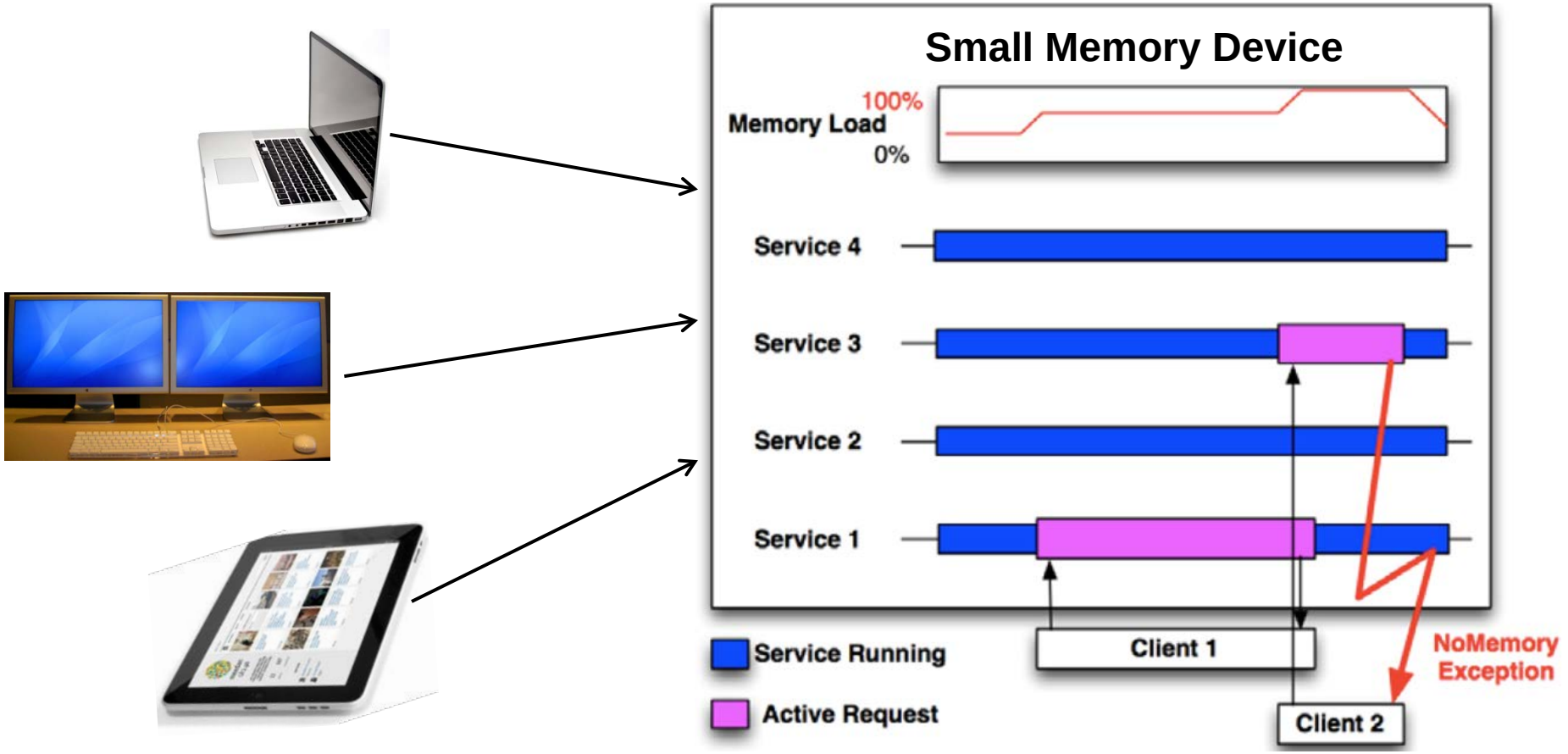




Add activator to the dynamic configuration path thru the JAWS pattern language

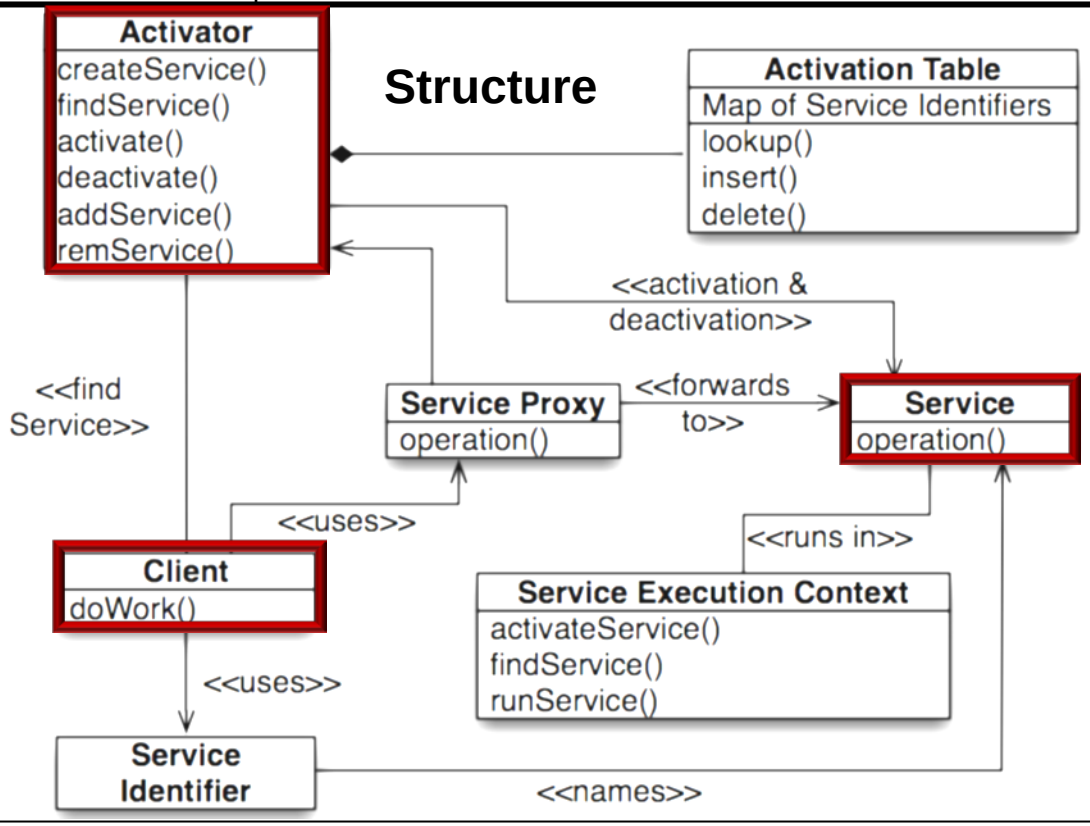
Minimizing Resource Utilization

Context	Problem
• Resource constrained & highly dynamic environments	• It may not feasible to have all application server implementations running all the time since this ties up end-system resources unnecessarily



Minimizing Resource Utilization

Context	Problem	Solution
• Resource constrained & highly dynamic environments	• It may not feasible to have all application server implementations running all the time since this ties up end-system resources unnecessarily	• Apply the ***Activator*** pattern to activate & deactivate JAWS web server automatically

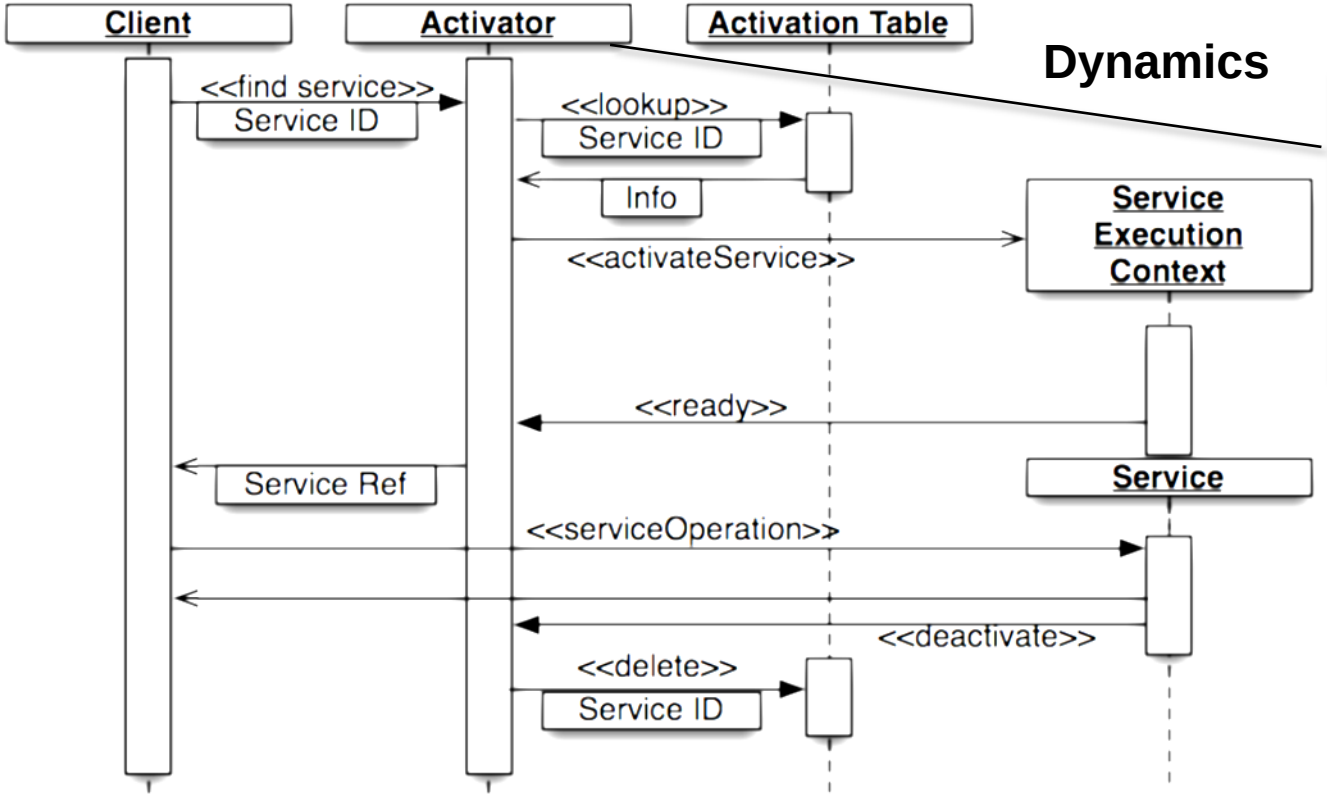


Activator automates scalable on-demand activation & deactivation of service execution contexts to run services accessed by many clients without consuming excessive resources

See www.dre.vanderbilt.edu/~schmidt/PDF/Activator.pdf for info on Activator

Minimizing Resource Utilization

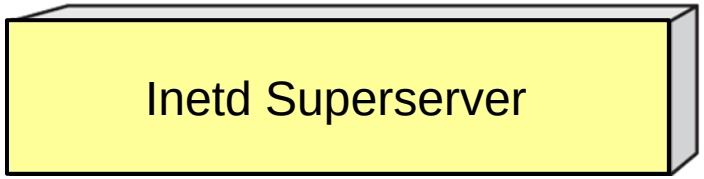
Context	Problem	Solution
• Resource constrained & highly dynamic environments	• It may not feasible to have all application server implementations running all the time since this ties up end-system resources unnecessarily	• Apply the ***Activator*** pattern to activate & deactivate JAWS web server automatically



An Activator can activate & passivate a service running in a server after each method call, each transaction, etc

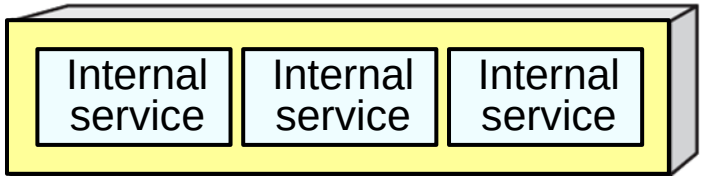
Applying Activator to JAWS

- *Activator* is supported the UNIX Inetd "superserver"



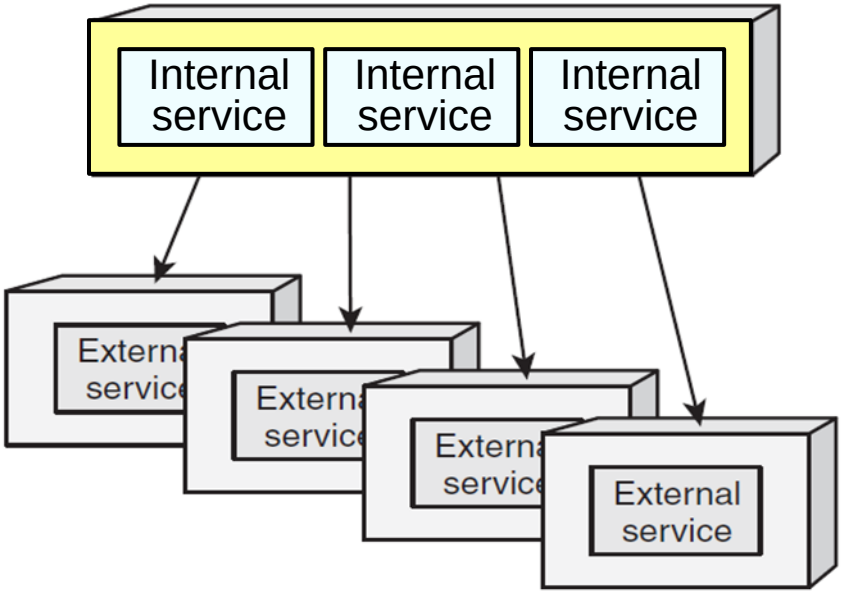
Applying Activator to JAWS

- *Activator* is supported the UNIX Inetd "superserver"
  - Internal services are fixed at static link time
    - e.g., **`ECHO`** & **`DAYTIME`**



Applying Activator to JAWS

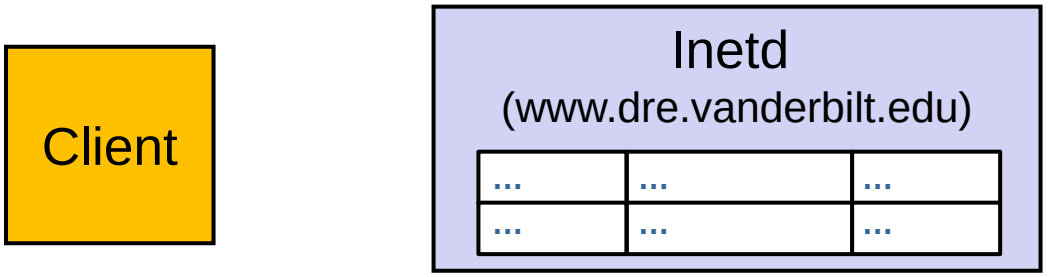
- *Activator* is supported the UNIX Inetd "superserver"
  - Internal services are fixed at static link time
    - e.g., **`ECHO`** & **`DAYTIME`**
  - External services can be dynamically reconfigured via sending a **`SIGHUP`** signal to Inetd, which then performs the **`socket()`**/**`bind()`**/**`listen()`** calls on all services listed in the **`inetd.conf`** file
    - e.g., **`FTP`**, **`TELNET`**, & **`HTTP`**



See en.wikipedia.org/wiki/Inetd for more on Inetd

Applying Activator to JAWS

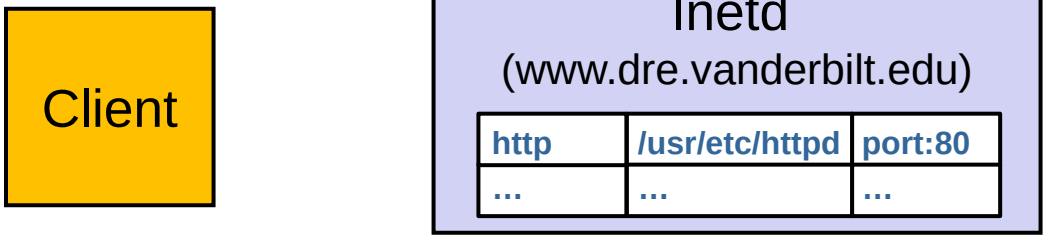
- Clients can use the Inetd-based *Activator* pattern to launch a JAWS-based web server on-demand



Applying Activator to JAWS

- Clients can use the Inetd-based *Activator* pattern to launch a JAWS-based web server on-demand
- Put following line in the **`/etc/inetd.conf`** file:

```
http stream tcp
  nowait root
  /usr/etc/httpd
  httpd
```

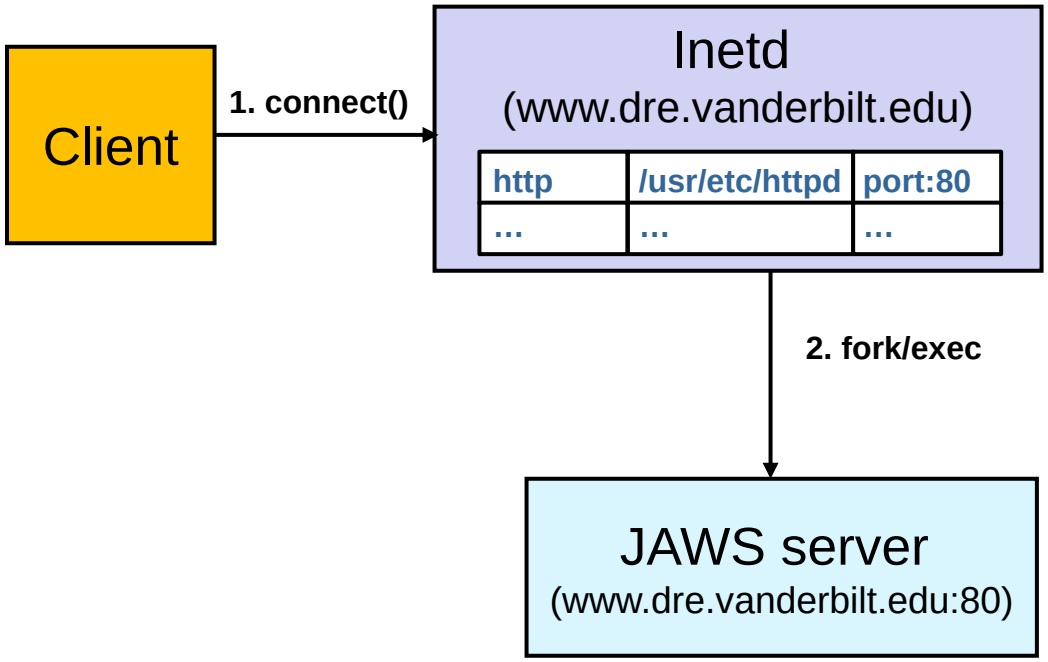


Applying Activator to JAWS

- Clients can use the Inetd-based *Activator* pattern to launch a JAWS-based web server on-demand
- Put following line in the `/etc/inetd.conf` file:

```
http stream tcp
  nowait root
  /usr/etc/httpd
  httpd
```

- When a TCP connection arrives on port 80, Inetd launches the JAWS-based server program

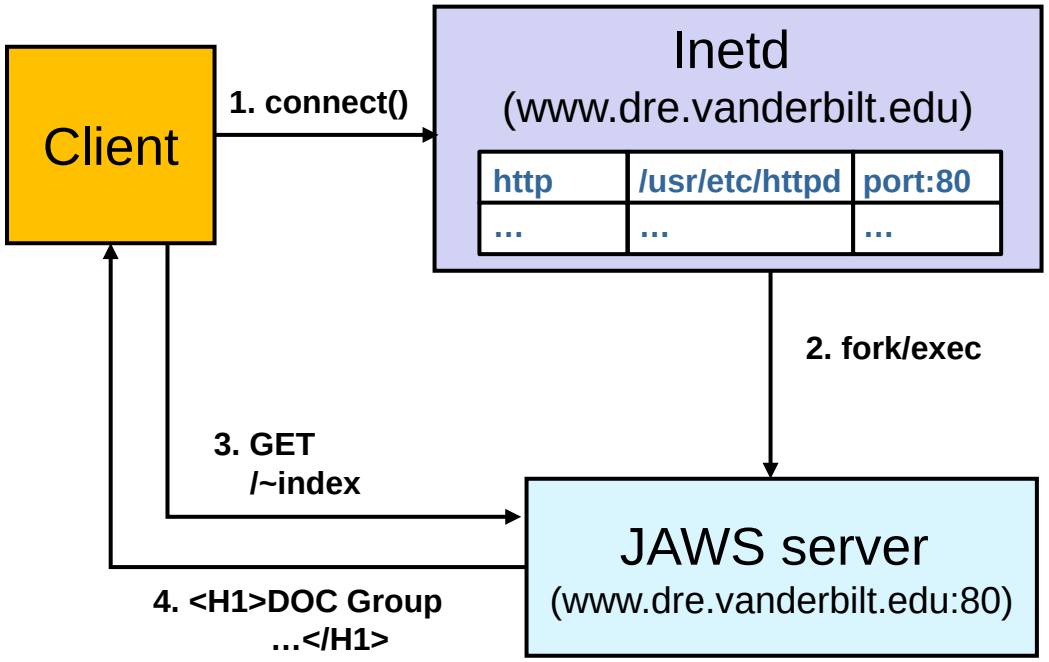


Applying Activator to JAWS

- Clients can use the Inetd-based *Activator* pattern to launch a JAWS-based web server on-demand
- Put following line in the `/etc/inetd.conf` file:

```
http stream tcp
  nowait root
  /usr/etc/httpd
  httpd
```

- When a TCP connection arrives on port 80, Inetd launches the JAWS-based server program
- This server then handles the client request(s)

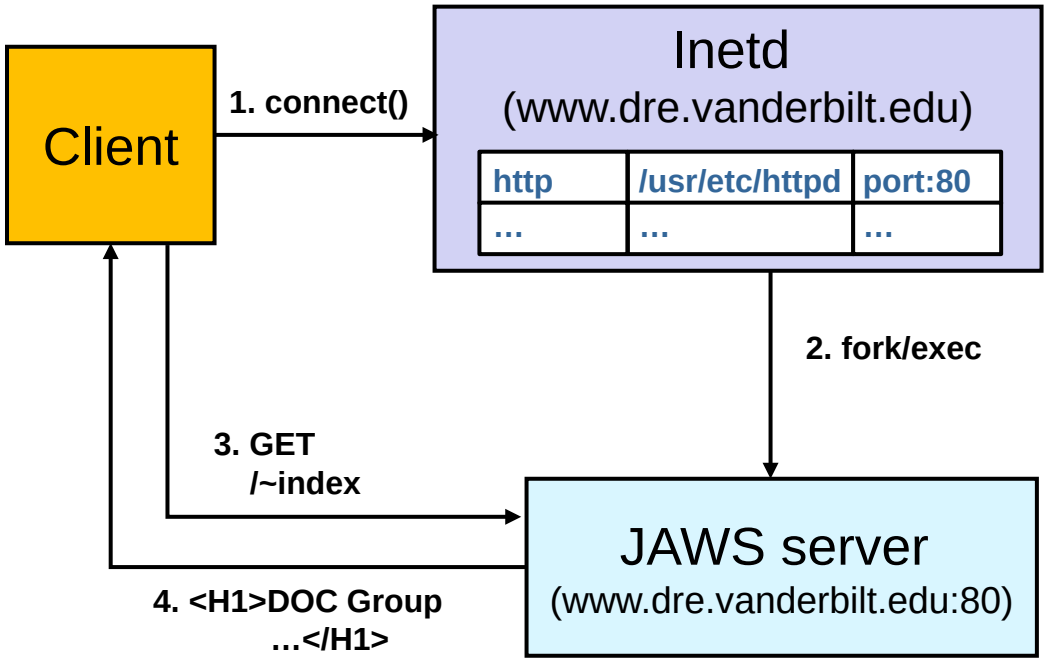


Applying Activator to JAWS

- Clients can use the Inetd-based *Activator* pattern to launch a JAWS-based web server on-demand
- Put following line in the `/etc/inetd.conf` file:

```
http stream tcp
  nowait root
  /usr/etc/httpd
  httpd
```

- When a TCP connection arrives on port 80, Inetd launches the JAWS-based server program
- This server then handles the client request(s)

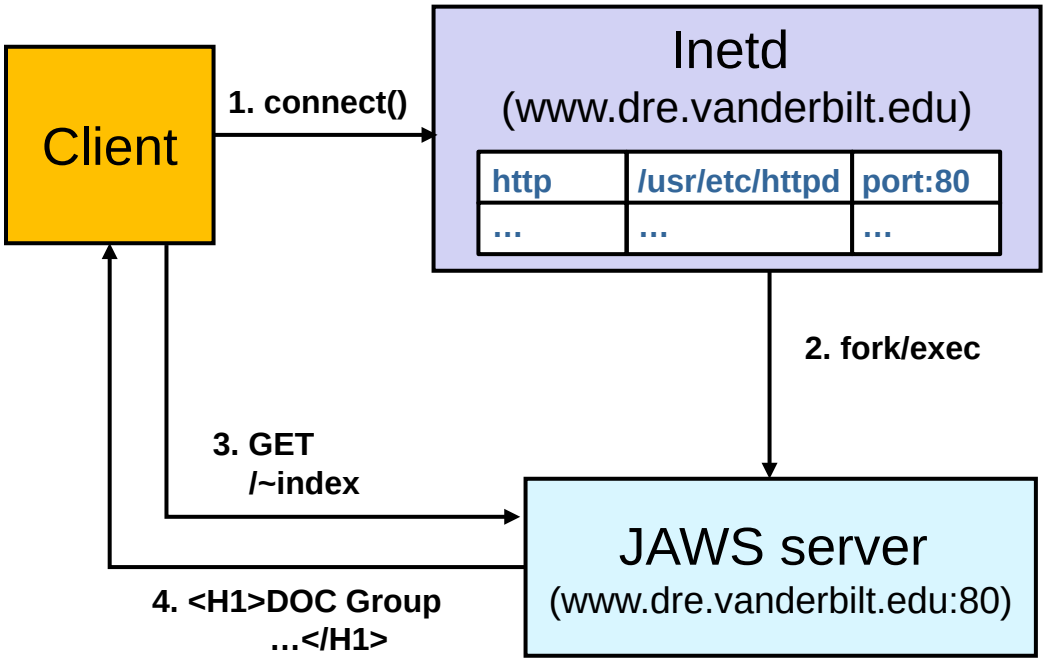


- This design uses memory more efficiently since the server runs only when needed
  - Appropriate for a web server that isn't expected to run with high loads

Benefits of the Activator Pattern

More effective resource utilization

- Servers can be spawned "on-demand," thereby minimizing resource utilization until clients actually require them



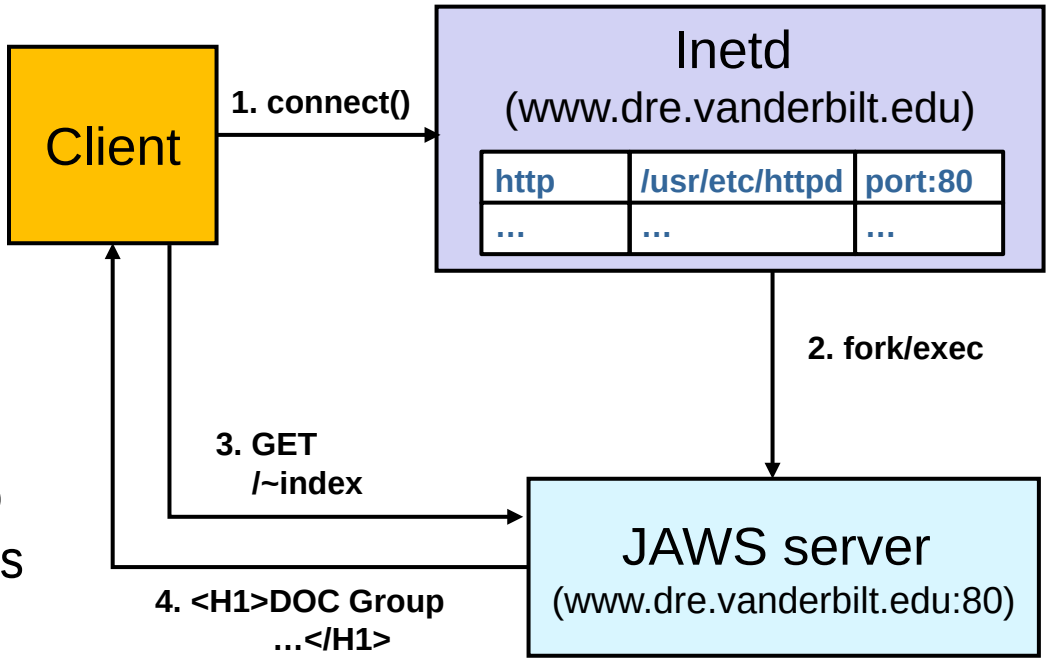
Benefits of the Activator Pattern

More effective resource utilization

- Servers can be spawned "on-demand," thereby minimizing resource utilization until clients actually require them

Coarse-grained concurrency

- By spawning server processes to run on multi-core/CPU computers



Benefits of the Activator Pattern

More effective resource utilization

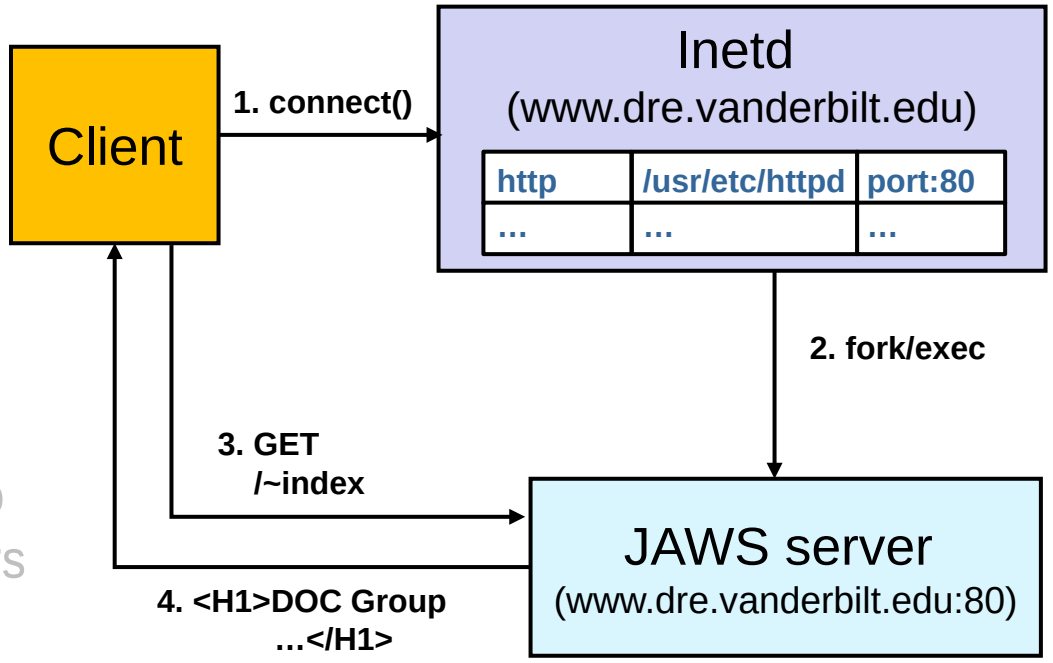
- Servers can be spawned "on-demand," thereby minimizing resource utilization until clients actually require them

Coarse-grained concurrency

- By spawning server processes to run on multi-core/CPU computers

Modularity, testability, & reusability

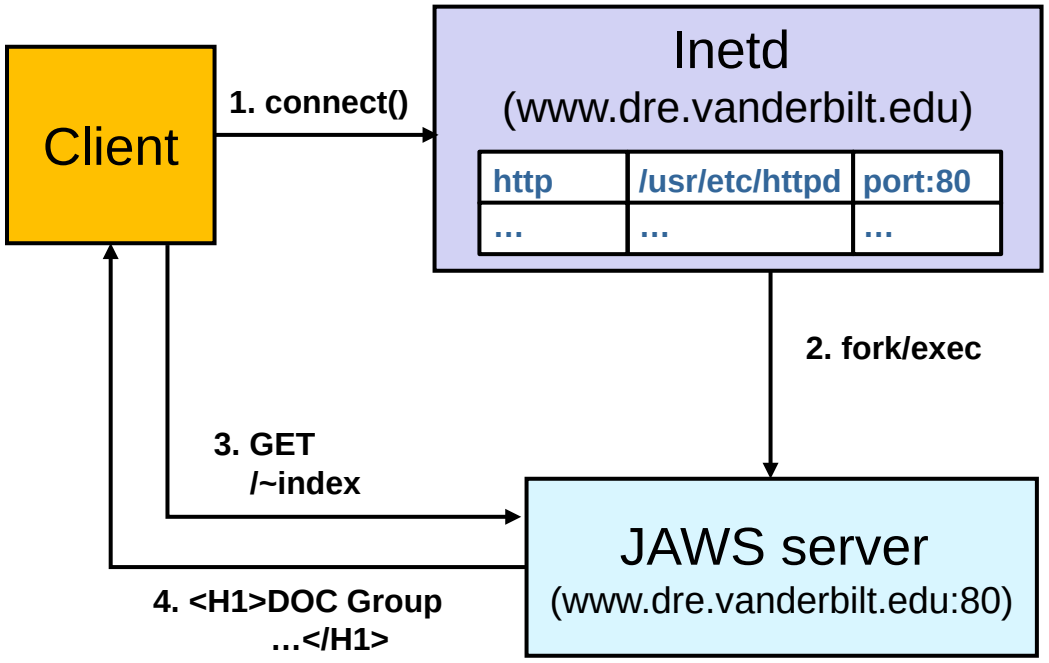
- Application modularity & reusability is improved by decoupling server implementations from the manner in which the servers are activated



Limitations of the Activator Pattern

Lack of determinism & ordering dependencies

- Hard to determine or analyze the behavior of an app until its components are activated at runtime



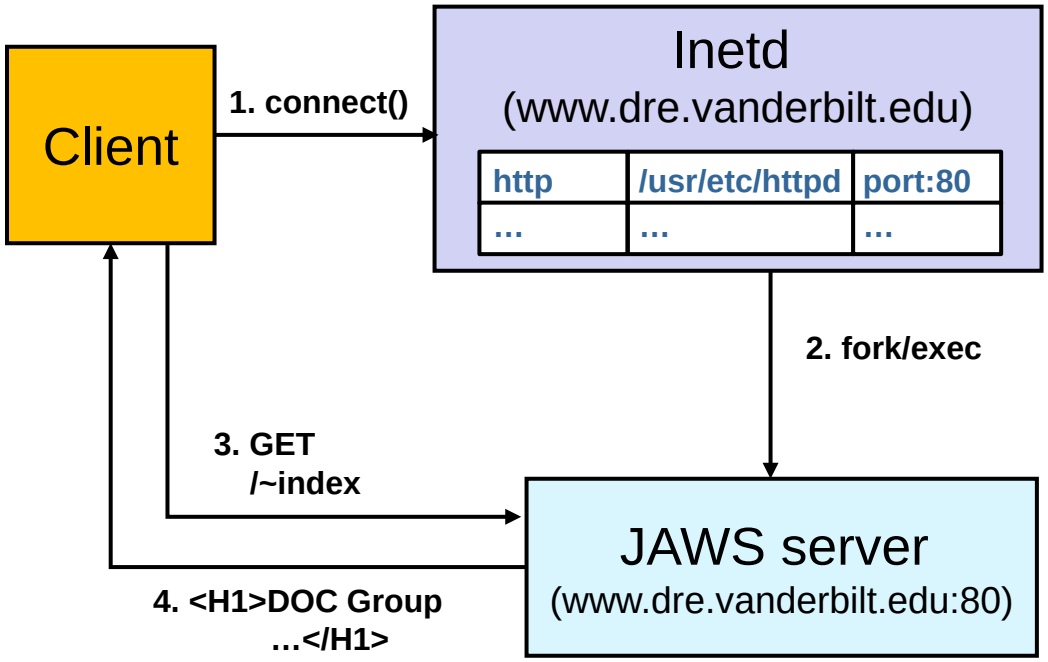
Limitations of the Activator Pattern

Lack of determinism & ordering dependencies

- Hard to determine or analyze the behavior of an app until its components are activated at runtime

Reduced security or reliability

- An application that uses *Activator* may be less secure or reliable than an equivalent statically-configured application



Limitations of the Activator Pattern

Lack of determinism & ordering dependencies

- Hard to determine or analyze the behavior of an app until its components are activated at runtime

Reduced security or reliability

- An application that uses *Activator* may be less secure or reliable than an equivalent statically-configured application

Increased run-time overhead & infrastructure complexity

- By adding levels of abstraction & indirection when activating & executing components