Motivation for Dynamic Architectures

- Benefits of architecture-based development are well understood
  - particular benefit in large-scale development
- Safety- and mission-critical software systems often cannot be shut down and restarted for upgrades
  - undesirable delays
  - increased cost
  - unacceptable risk
  - air traffic control
  - telephone switching
  - 24x7 public information systems,
- Runtime modification is a key aspect of these systems
  - this support should be provided at the level of architecture

What Are “Dynamic Architectures?”

- Architecture is not an executable artifact
- Two possibilities
  - *simulate* system execution at the architectural level
  - reflect architecture modifications in *executing system*
- Support for architectural dynamism
  - constrained dynamism
    - all changes to the architecture must be known *a priori*
  - unconstrained architectural dynamism
    - any changes are allowed in principle
    - the validity of the changes must be ensured at runtime
Aspects of Dynamic Software Architectures

- Modeling dynamic architectures
  - languages for describing dynamically evolving software architectures
  - enable runtime changes

- Specifying dynamic changes
  - architecture modification languages
  - specify runtime changes

- Governing change
  - restricting runtime change to preserve system integrity

- Runtime tool support
  - tools for constructing dynamic systems from architectures
  - tools for supporting runtime changes

Modeling Dynamism in Architectures

- ADLs that support dynamism
  - most support constrained dynamism
  - C2 and Weaves support unconstrained dynamism

- Approaches to constrained dynamism
  - parameterized instantiation of architectural elements
  - component/connector replication
  - conditional reconnection
  - special architectural modification events

- Approaches to unconstrained dynamism
  - addition
  - removal
  - replacement
  - reconnection
  - requires real-time analysis
Constrained Dynamism

- ADLs provide modeling support
  → enactment requires tools
- Example — Darwin
  - runtime component replication via dynamic instantiation
    ```
    provide create_inst<\text{dyn} \text{data}>;
    bind create_inst -- \text{dyn} \text{comp};
    ```
  - component deletion/rebinding by interpreting scripts
- Example — Rapide
  - runtime reconnection via conditional operators
    ```
    \text{connections}
    \text{?A}: \text{Airplane}; \quad \text{?M}: \text{Msg};
    \text{SFO}: \text{Control\_Center};
    
    \text{?A.Radio(?M) Where ?A.InRange(SFO)}
    ||> \text{SFO.Receive(?M)};;
    ```

Unconstrained Dynamism

- ADLs’ underlying semantics provide support
  → enactment requires tools
  → requires on-the-fly analysis
- Achievable by
  - minimal element interdependencies
  - flexible connectors
  - untyped data
- Example — C2

```
```
Specifying Dynamic Changes

- ADLs describe architectures
  - used for analysis and system generation
  - include planned dynamic behavior

- Architecture modification languages (AMLs) describe changes to architectural descriptions
  - introduce unplanned changes to deployed systems
  - require active agent facilities to effect architectural changes in the system
    - difficult task in general
    - more tractable in implementation constraining approaches

→ An entire architectural description may be produced with an AML

AML Example

```
newArchitecture(A1);
A1.addComponent(C1);
A1.addComponent(C2);
A1.addComponent(C3);
A1.addConnector(B1);
A1.addConnector(B2);
A1.attach(C1,B1);
A1.attach(B1,C2);
A1.attach(C2,B2);
A1.attach(B2,C3);
A1.attach(B1,B2);
A1.addComponent(C4);
A1.addConnector(B3);
A1.attach(B3,C4);
A1.detach(B1,B2);
A1.attach(C2,B3);
A1.detach(C2,B2);
A1.removeComponent(C3);
A1.removeConnector(B2);
```
Governing Dynamic Architectural Changes

- Placing constraints on allowed changes
  - structural
  - syntactic
  - semantic

- Numerous possibilities for specifying constraints
  - imperative via predicates
    - e.g., Require (connect A B)
      Prohibit (connect A C)
  - declarative
    - e.g., Forall c1,c2 in sys.Comps @ Exists b in sys.Conns @
      Attached(c1,b) and Attached(c2,b)
  - event patterns
  - communication protocols

Tool Support for Dynamism

- Tools to facilitate use of dynamic software architectures
  - APIs, event mechanisms

- Construction of systems that exhibit dynamic properties

- Enactment of AML-like commands in running systems

- Analysis of dynamic changes
  - change the model, analyze it, enact changes in the system

- Generation of code needed to effect dynamic changes
  - glue code, new connectors, new components

- Provision of mechanisms that provide low-level support for dynamism
  - e.g., access to OS services, DLLs, PL-specific dynamic features
ArchShell
A Tool for Supporting Architectural Dynamism

- Allows C2-style architectures to be constructed, executed, monitored, and modified interactively
- Implements AML commands for component/connector
  - addition — addComponent, addConnector
  - removal — removeComponent, removeConnector
  - (re)connection — weld, unweld
- Built on top of the C2 implementation infrastructure
  - depends on the infrastructure to enable dynamic changes
- Similar to a UNIX command shell (e.g., csh)
  - allows interactive system construction
- Different from a UNIX shell
  - supports architecture modification during runtime

C2 Implementation Infrastructure

- Simple, extensible framework of abstract classes for C2 concepts
  - components
  - connectors
  - communication ports
  - messages
- Implements interconnection and communication protocols
- Enables rapid construction of C2-style applications
  - allows developers to focus on application-level issues
- Implemented in Java, C++, and Ada (partially)
  - you will use the Java framework for Homework 3c
Java Framework — Message

- Messages parameters are Java hashtables
  - notification vs. request
- Methods
  - `name` — returns message name
  - `setName` — sets message name
  - `type` — returns message type
  - `setType` — sets message type
  - `addParameter` — adds parameter to message
  - `removeParameter` — removes parameter from message
  - `getParameter` — retrieves parameter value
  - `hasParameter` — checks if message has given parameter
  - `setAllParameters` — sets all message parameters at once
  - `getAllParameters` — returns parameter hashtable
  - `clone`
Java Framework — Port

- Abstract class
- Methods
  - `belongsTo` — returns brick to which port belongs or (re)assigns port to brick
  - `Link` — accesses port to which `this` is linked
  - `weld` — welds `this` to another port
- FIFOPort is a kind of port
- FIFOPort Methods
  - `receive` — add message to incoming queue and notify brick
  - `selectNextIncomingMessage` — from incoming queue
  - `selectNextOutgoingMessage` — from outgoing queue
  - `send` — remove message from outgoing queue and transfer to its link’s incoming queue

Java Framework — Brick

- Methods
  - `start` — starts the brick running; must be invoked first
  - `isStarted` — checks if brick has started
  - `finish` — immediately stop brick’s execution
  - `timeStep` — abstract method invoked periodically to allow the brick to execute
Java Framework — Component

- Methods
  - `bottomPort` — returns component’s bottom port
  - `topPort` — returns component’s top port
  - `send` — messages to components above and below
  - `timeStep` — process an incoming message from each port
  - `handle` — abstract methods invoked by `timeStep` for handling notifications and requests

- Declaring new components
  - inherit from `Component`
  - define `handle` methods

- Using new components
  - create an instance
  - call its `start` method

Java Framework — Connector

- C2 connectors have `context-reflective` interfaces
  - `add` `Top` `add` `Bottom` `Port`
  - `remove` `Top` `Bottom` `Port`

- Methods
  - `topPortAt` — returns top port at specified index
  - `bottomPortAt` — returns bottom port at specified index
  - `topPorts` — returns list of connector’s top ports
  - `bottomPorts` — returns list connector’s bottom ports
  - `handle` — methods for processing requests and notifications
  - `send` — broadcast message to all bricks above or below `this`
  - `timeStep` — process an incoming message from each port

- Declaring new connectors
  - inherit from `Connector`
  - override `handle` methods — message filtering
  - override `send` methods — message routing
Java Framework — [Brick]Thread

- Each brick runs in its own thread of control
- Implemented with a semaphore
- Methods
  - `start` — starts the thread running
  - `finish` — stops the execution of the thread
  - `finished` — checks if the brick is idle
  - `run` — called when thread created; processes messages or waits for them
  - `newMessage` — invoked when brick receives new message; sets semaphore to wake up brick’s thread

Java Framework — Architecture

- Methods
  - `start` — starts all bricks in the architecture
  - `run` — assigns `timeSteps` to all bricks in the architecture
  - `finish` — finishes all bricks in the architecture
  - `finished` — checks if all bricks have finished executing
  - `add[Brick]` — adds brick to architecture
  - `remove[Brick]` — removes brick from architecture
  - `Components` — returns vector of components in `this`
  - `Connectors` — returns vector of connectors in `this`
  - `detach[Side]` — detaches all attached bricks on given side
  - `detach[Brick]` — detaches all bricks from both sides
  - `weld` — attaches two or more bricks to each other
  - `unweld` — detaches two or more bricks from each other
C2 Graphics Binding

- Existing toolkits essentially issue notifications...
  - “the user has typed the ‘x’ key”
- ...and respond to requests
  - “draw a line from \((x_1, y_1)\) to \((x_2, y_2)\)”
- Toolkits conceptually at the bottom of a C2 architecture
  - receive notifications
  - issue requests
- A C2 graphics binding is an *adaptor* to the toolkit

Java Framework — *GraphicsBinding*

- Subclass of *ComponentThread*
  - C2 notifications \(\rightarrow\) AWT calls
  - user-generated events \(\rightarrow\) C2 requests
- Current support
  - frames
  - text fields
  - rectangles
  - panels
  - arcs
  - text strings
  - lists
  - lines
  - buttons
  - ovals
- Each supported element has an associated “C2” class
  - may or may not extend AWT class
  - exports needed interface
  - can be extended
- No AWT layout managers are used
**GraphicsBinding Methods (1)**

- **handle** — handles C2 notifications received from above
- **send** — sends C2 requests up the architecture
- **CreateObject** — creates screen object
- **ModifyObject** — modifies screen object
- **DestroyObject** — destroys screen object
- **createViewport** — creates C2 viewport (AWT Frame)
- **destroyViewport** — destroys C2 viewport (AWT Frame)
- **clearViewport** — clears C2 viewport
- **addPanel** — creates C2 panel (AWT Panel)
- **clearPanel** — clears C2 panel

**GraphicsBinding Methods (2)**

- **addList** — adds list to container
- **appendListItem** — appends item to end of list
- **addListItem** — adds item to list at given location
- **removeListItem** — remove item at given location from list
- **replaceListItem** — replaces item at given location with new item
- **(de)selectListItem** — marks list item as (de)selected
- **clearListItems** — clears all items from list
- **addTextField** — adds text field to container
- **eraseTextField** — clears text field
- **setTextField** — sets text field’s value
- **getTextField** — retrieves text field’s value
- **addButton** — adds button to container (panel or viewport)
**GraphicsBinding Methods (3)**

- **line** — draws a line in container
- **arc** — draws arc in container
- **oval** — draws oval in container
- **rectangle** — draws rectangle in container
- **text** — draws text string in container
- **shutdown** — exits current application

**Notification Handling in GraphicsBinding**

```java
if (message_name.equalsIgnoreCase("ButtonAdded"))
{
    int x = ((Integer)n.getParameter("x")).intValue();
    int y = ((Integer)n.getParameter("y")).intValue();
    int width = ((Integer)n.getParameter("width")).intValue();
    int height = ((Integer)n.getParameter("height")).intValue();
    Color foreground = toColor((String)n.getParameter("foreground"));
    Color background = toColor((String)n.getParameter("background"));
    String id = (String)n.getParameter("label");
    String parent_id = (String)n.getParameter("parent_id");

    Container c = containerByID(parent_id);
    if (c.getClass().getName().equals("c2.comp.graphics.C2Viewport"))
        addButton((C2Viewport)c, x, y, width, height, id, foreground, background);
    else
        addButton((C2Panel)c, x, y, width, height, id, foreground, background);
}
```
Extending *GraphicsBinding*

- Define or modify “C2” class associated with (new) graphics element
  - constructors
  - accessors
  - modifiers
- Expand C2Viewport and C2Panel classes
  - events and actions needed by new element
  - requests sent up the architecture
- Expand the *GraphicsBinding handle* method
  - notifications received to operate on new element
  - element’s C2 class methods invoked in response