CS 351
Design of Large Programs
Architectural Design Patterns

Brooke Chenoweth

University of New Mexico

Spring 2025

Software Development Revisited

1. Specification
    - precisely define the problem to be solved
    - validate one's understanding of the problem
2. Design
    - outline a solution path
    - plan the implementation
3. Implementation
    - build the software
    - use the constructs available in the programming language

Implications for this Course

Skill development with a focus on design and implementation
Specific skills to be acquired:

- ability to understand and conceptualize a problem
- ability to lay out a coherent and complete design that solves the problem
- ability to plan an implementation targeted to a specific programming language
- ability to deliver fully functioning code incrementally

Pragmatic and systematic application of agile programming approach

- weekly delivery of functioning versions of the program under development

Is Our Perspective Unique?

Specification/Design/Implementation paradigm is not specific to software

- kitchen
- landscape
- electronic device
- mechanical device

In software engineering as well as in other engineering disciplines, the specification/design cycle is applied recursively

- system
- subsystem
- component
- subcomponent

Software Architecture

The design of a software system is captured by a *Software Architecture Design*

- an abstract description of the system's structure and behavior
- not an exact reflection of the code organization

The level of abstraction is chosen such that:

- all critical design decisions are apparent
- meaningful analysis is feasible
- implementation plans can be developed
- all interfaces are precisely defined

Why Bother?

No major engineering achievement is possible without design and analysis

- home building without plans
- car manufacturing without precise part specifications
- radiation treatment machine without precise analysis
- moon shot by trial and error

Teamwork demands a common plan of action and coordination

Design Diagram

A typical software architecture is specified by a combination of:

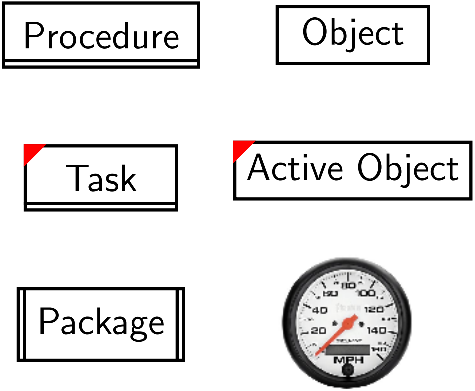
- design diagrams
- component specifications
- external interface specifications

A design consists of two types of entities:

- *components* – code modules relevant to the overall design
- *connectors* – suggestive of the interactions among components

Notation: Components

- Passive
  - procedure
  - object
- Active
  - task
  - active object
- Organizational
  - package
- External
  - devices and interfaces



Notation: Connectors

Architecture diagrams may use a wide range of connector types:

- standard (widely used in the literature)
- custom (defined specifically to meet the needs of a particular system)

Basic connectors:

- *aggregation* – structural abstraction
- *reference* – behavioral constraint

Connector: Aggregation

The *aggregation* connector captures structural properties of objects

- constrains the scope of object definitions
- constrains the method invocation pattern

Connector: Aggregation

Aggregation is a relation between

- an object and lower level objects to which is has exclusive access
- the scope of the subordinate objects is limited to the object above
- subordinate objects are often instances of some general class
    - may have an independent existence
    - may be used in different settings

Aggregation makes object composition possible

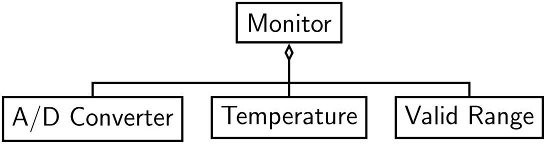
Illustration: Aggregation

```java
public class Monitor {
  private Temperature temp;
  private ADConverter converter;
  private ValidRange range;

  public void update() {
    // Updates monitor with current reading from the ADConverter.
  }

  public void setMinValue(Temperature temp) {}
  public void setMaxValue(Temperature temp) {}
  public boolean inRange() {}

  public void clearHistory() {
    // Clears the list of readings that are out of range.
  }
}
```



Connector: Reference

The *reference* connector captures run time object usage pattern

- constrains the method invocation pattern

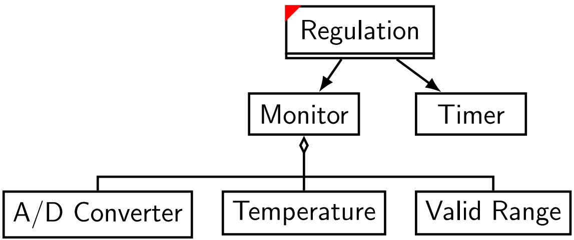
Reference is a relation between:

- a procedure and the objects it accesses
- an object and lower level objects it accesses

Illustration: Reference

```
public void regulateTemp() {
  long updateInterval = 500;
  timer.setInterval(updateInterval);
  while(!timer.timedOut()) {
    monitor.update();

    if(!monitor.inRange()) {
      // ...
    }
    // sleep(10) ...
  }
}
```



Static vs. Dynamic Systems

A system is *static* in nature if its structure does not evolve at runtime

- design diagrams are also static in nature – a good match

A system whose structure evolves during runtime execution is *dynamic*

- new components are created
- connector patterns change

Static Diagrams for Dynamic Systems

The use of static design diagrams is made more difficult when designing a dynamic system

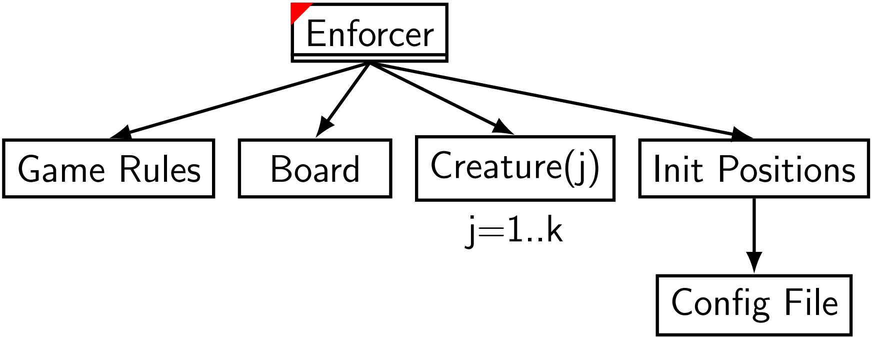
- capture the most representative structure statically
- capture one or more representative structures
- explain the system evolution rules separately

A Static System

Consider a board game called LiveChess:

- standard chess board
- pieces are creatures with a mind of their own
- a fixed set of pieces are used
- a configuration file defines the initial placement of pieces
- pieces are given turns to move according to some set of rules
- each piece selects a move which is executed only if valid

A Static System

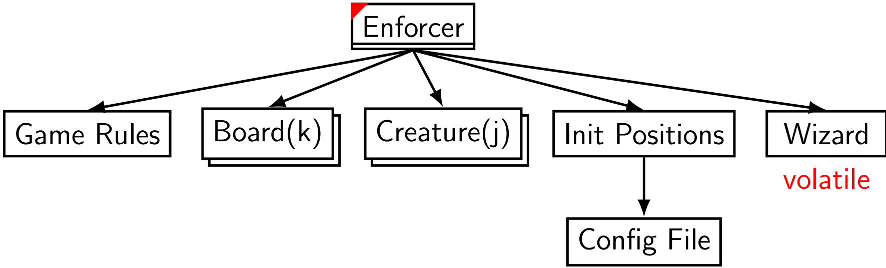


A Dynamic System

Consider a new version of LiveChess:

- the board may change in size over time – *no impact on the diagram*
- creatures may be born and may die – *variable set of objects*
- new worlds may be created as additional board games – *variable set of objects*
- a wizard may materialize from time to time – *typical configuration should include it*

A Dynamic System



Architectural Patterns

An *architectural pattern* may be defined as a generic design which

- has some desirable property
- solves some frequently encountered problem
- offers a good starting point for a solution
- provides a reusable structure applicable to some problem domain

Meta-level considerations are not immediately explicit in the structure alone – they may be need to be considered

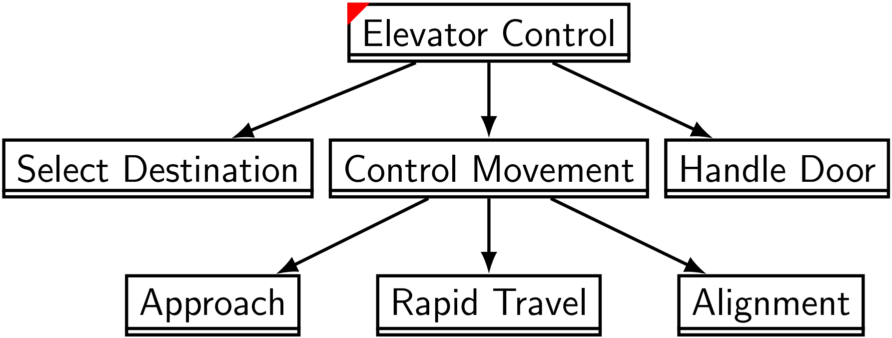
The basic object-oriented design is such a pattern. When used properly it promotes:

- information hiding
- encapsulation

Functional Decomposition

- Functional decomposition may be employed in order to encapsulate policy decisions and to control the complexity of:
    - the processing logic
    - non-trivial methods
- The relation defining the interactions among procedures is a *reference*, which constrains who can invoke whom

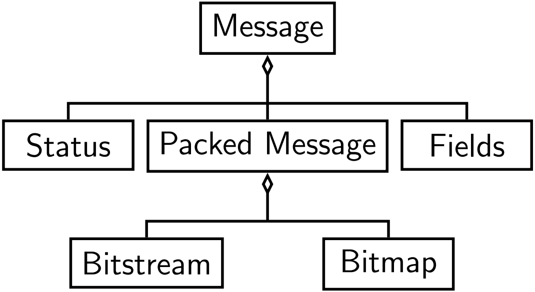
Functional Decomposition Example



Nested Objects

Nested objects are constructed strictly through the use of aggregation (tree structure)

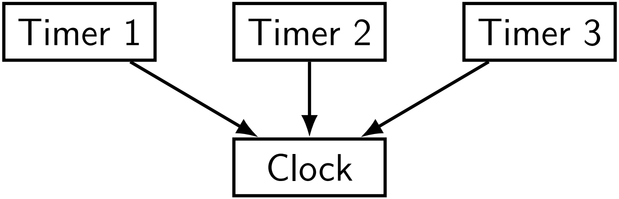
- each object can reference only its subcomponents
- it is desirable for sibling objects to be of similar complexity and level of abstraction



Shared Resources

Object sharing is highly undesirable

When sharing cannot be avoided it should be minimized, structured, and made uniform



Transparent Sharing

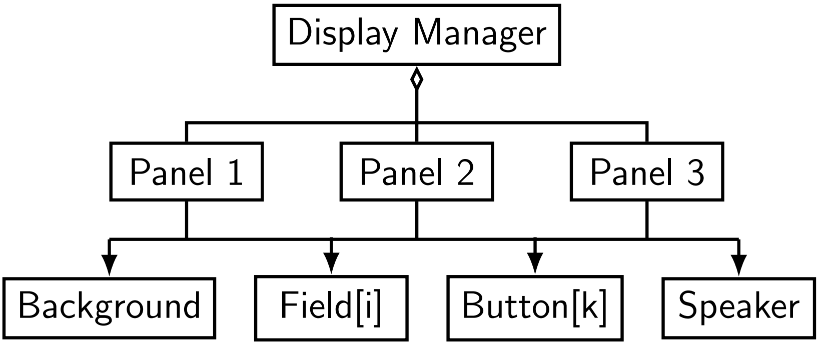
- Object sharing occurs when two or more objects have an acquaintance (reference) in common
- Transparent sharing occurs when none of the objects involved can detect that sharing takes place
- This is often the case when one physical interface supports several logical interfaces

Layered Objects

- A layered object consists of a hierarchically organized set of objects
- An object at one level can reference all objects on the level below
- Sharing is not transparent
- The level of abstraction decreases with depth

Layered object example

- Dynamic restructuring
- One panel is active at a time

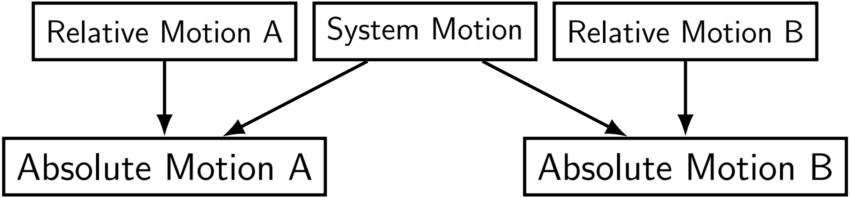


Mutation

- Mutation is an abstraction pattern that relates two object layers
- It involves a change in the encapsulation of the composite state of the lower level in response to the needs of the upper levels in the design
- It is especially helpful for restructuring low level physical interfaces into more abstract ones
- The sharing of the lower level objects must be transparent

Mutation Example

- Consider pair of objects in motion
- Know absolute motion, would prefer relative



Shared Implementation

- Performance considerations often require objects which are essentially independent to be encapsulated in a single object managing their implementation
- The desire for generality may also lead to shared implementations

Object Veneer

- Legacy code need not be an impediment in the application of object-oriented design
- Existing code can be encapsulated as a set of objects which are available to the remainder of the system