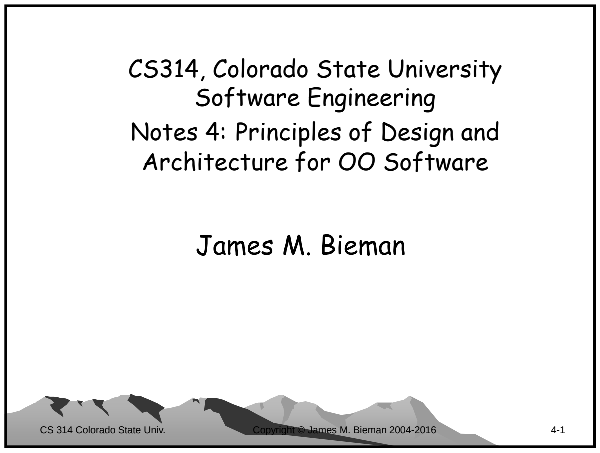
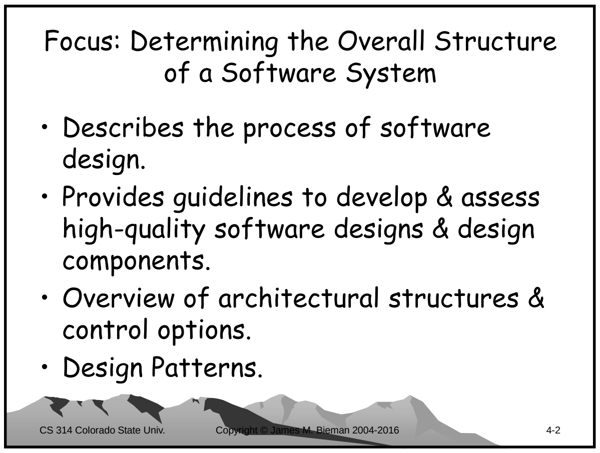


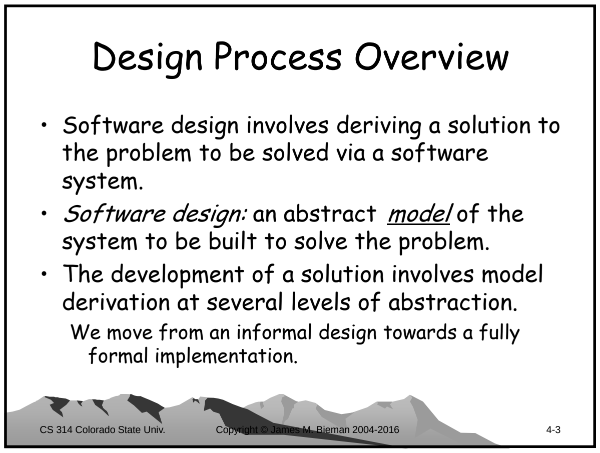
CS314, Colorado State University
Software Engineering
Notes 4: Principles of Design and Architecture for OO Software

James M. Bieman



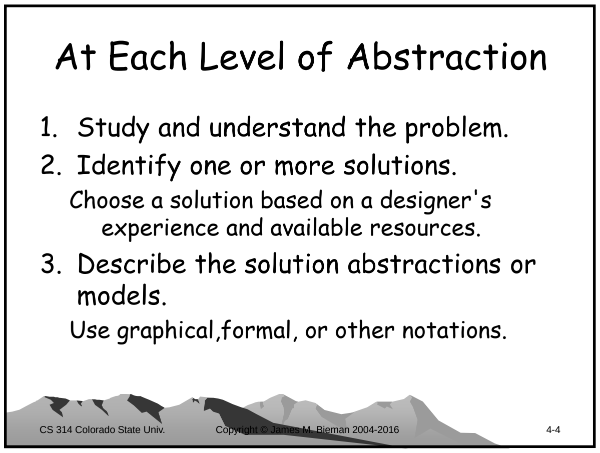
Focus: Determining the Overall Structure of a Software System

- Describes the process of software design.
- Provides guidelines to develop & assess high-quality software designs & design components.
- Overview of architectural structures & control options.
- Design Patterns.



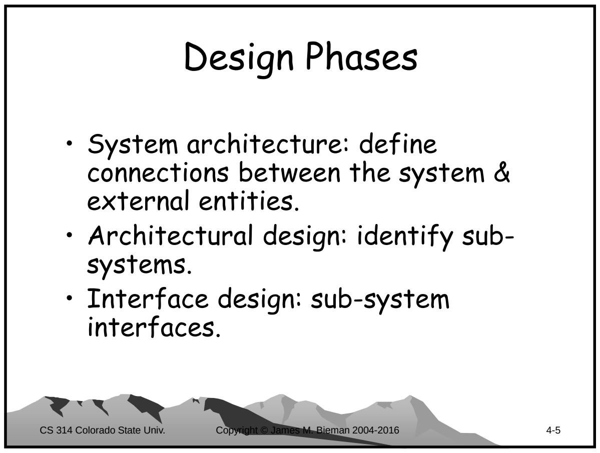
Design Process Overview

- Software design involves deriving a solution to the problem to be solved via a software system.
- *Software design:* an abstract *model* of the system to be built to solve the problem.
- The development of a solution involves model derivation at several levels of abstraction.
  We move from an informal design towards a fully formal implementation.



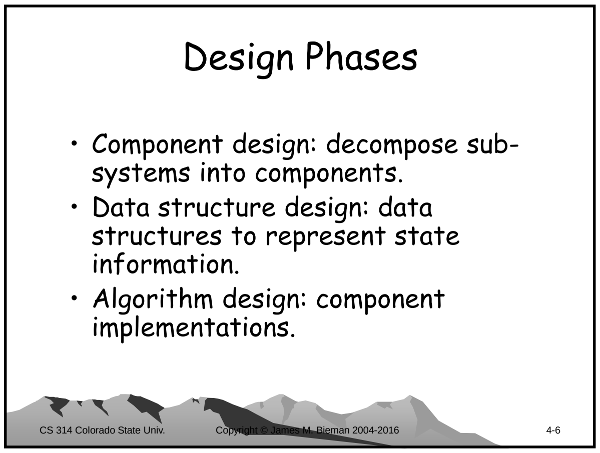
At Each Level of Abstraction

1. Study and understand the problem.
2. Identify one or more solutions.
   Choose a solution based on a designer's experience and available resources.
3. Describe the solution abstractions or models.
   Use graphical,formal, or other notations.



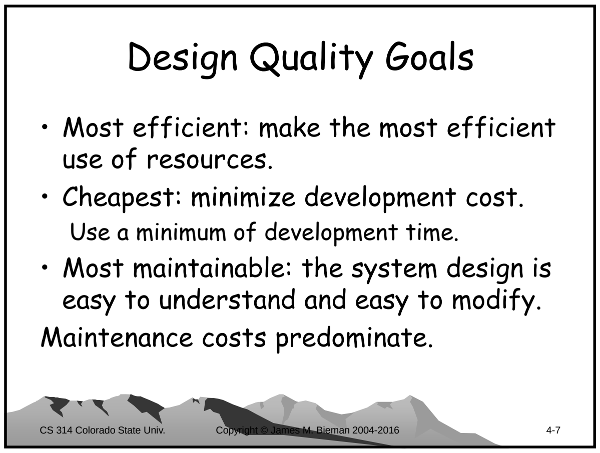
Design Phases

- System architecture: define connections between the system & external entities.
- Architectural design: identify sub-systems.
- Interface design: sub-system interfaces.



Design Phases

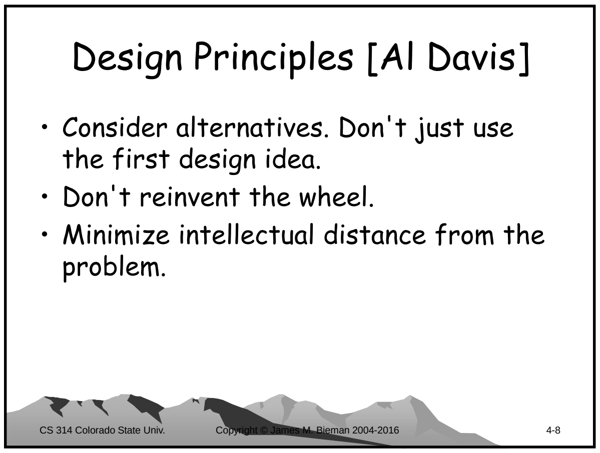
- Component design: decompose sub-systems into components.
- Data structure design: data structures to represent state information.
- Algorithm design: component implementations.



Design Quality Goals

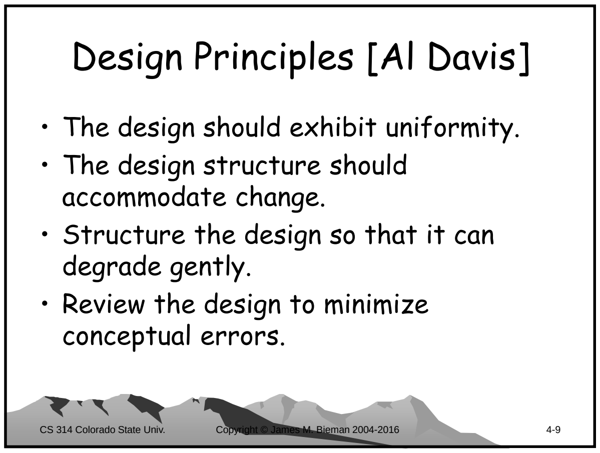
- Most efficient: make the most efficient use of resources.
- Cheapest: minimize development cost.
  Use a minimum of development time.
- Most maintainable: the system design is easy to understand and easy to modify.

Maintenance costs predominate.



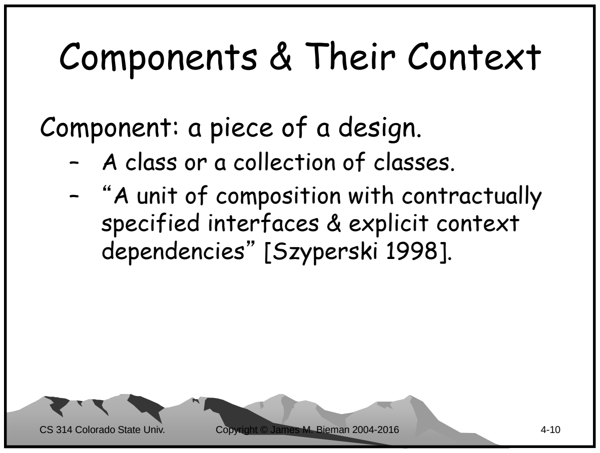
Design Principles [Al Davis]

- Consider alternatives. Don't just use the first design idea.
- Don't reinvent the wheel.
- Minimize intellectual distance from the problem.



Design Principles [Al Davis]

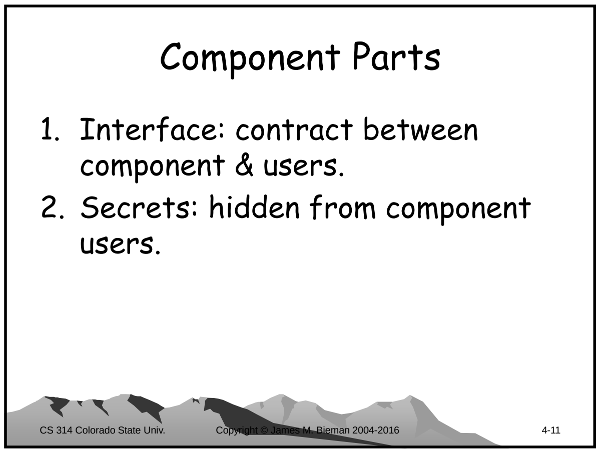
- The design should exhibit uniformity.
- The design structure should accommodate change.
- Structure the design so that it can degrade gently.
- Review the design to minimize conceptual errors.



Components & Their Context

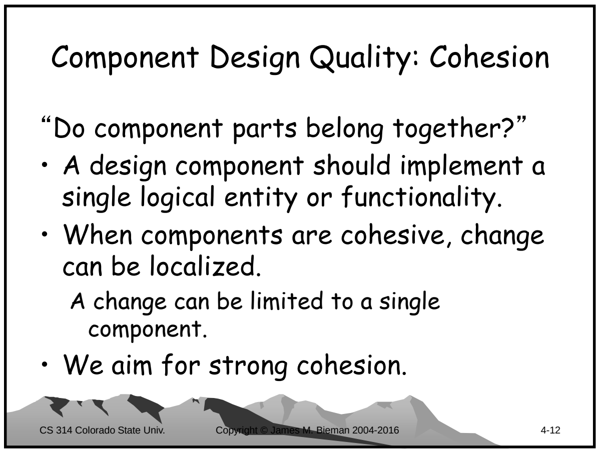
Component: a piece of a design.

- A class or a collection of classes.
- “A unit of composition with contractually specified interfaces & explicit context dependencies” [Szyperski 1998].



Component Parts

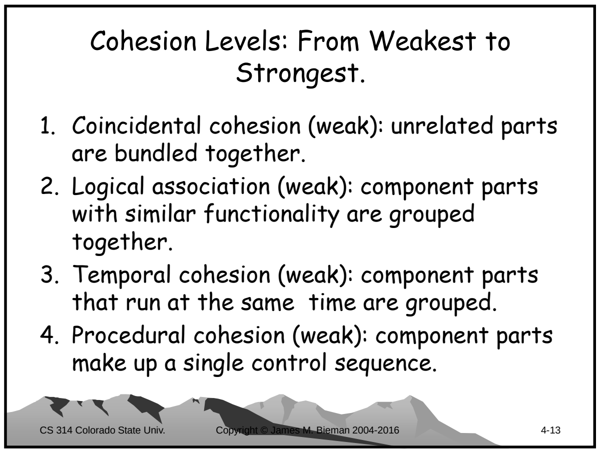
1. Interface: contract between component & users.
2. Secrets: hidden from component users.



Component Design Quality: Cohesion

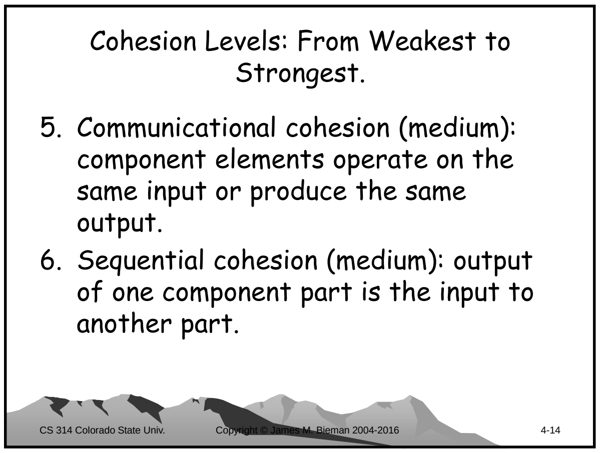
“Do component parts belong together?”

- A design component should implement a single logical entity or functionality.
- When components are cohesive, change can be localized.
  A change can be limited to a single component.
- We aim for strong cohesion.



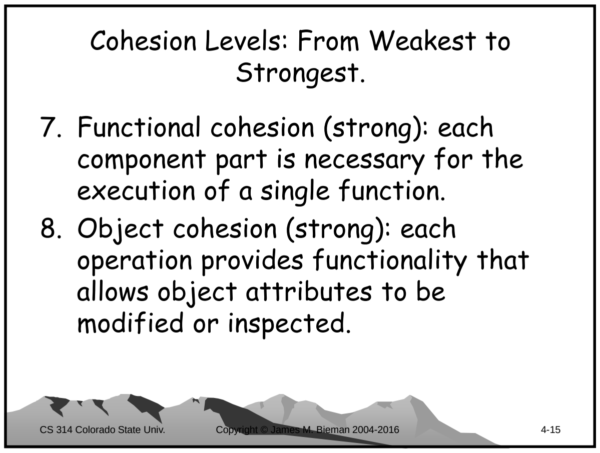
Cohesion Levels: From Weakest to Strongest.

1. Coincidental cohesion (weak): unrelated parts are bundled together.
2. Logical association (weak): component parts with similar functionality are grouped together.
3. Temporal cohesion (weak): component parts that run at the same time are grouped.
4. Procedural cohesion (weak): component parts make up a single control sequence.



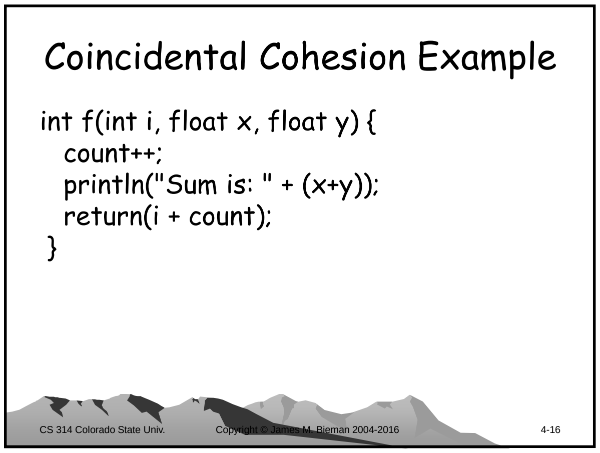
Cohesion Levels: From Weakest to Strongest.

5. Communicational cohesion (medium): component elements operate on the same input or produce the same output.
6. Sequential cohesion (medium): output of one component part is the input to another part.



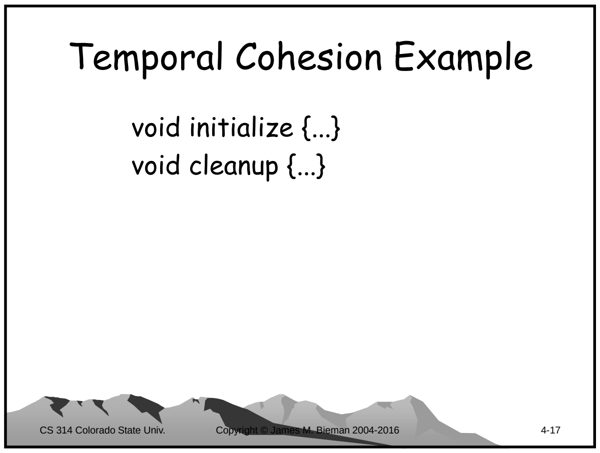
Cohesion Levels: From Weakest to Strongest.

7. Functional cohesion (strong): each component part is necessary for the execution of a single function.
8. Object cohesion (strong): each operation provides functionality that allows object attributes to be modified or inspected.



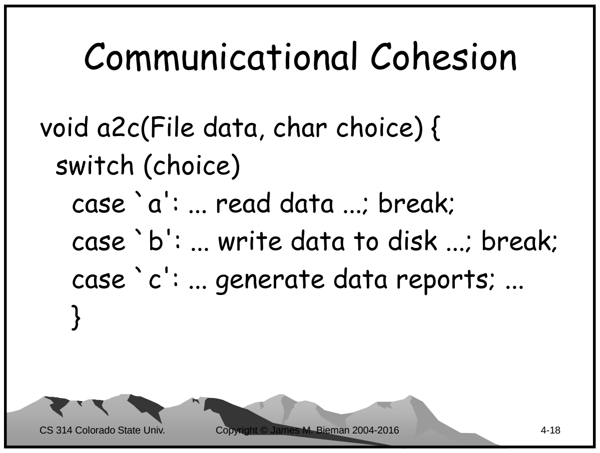
Coincidental Cohesion Example

```
int f(int i, float x, float y) {
  count++;
  println("Sum is: " + (x+y));
  return(i + count);
 }
```



Temporal Cohesion Example

```
void initialize {...}
void cleanup {...}
```



Communicational Cohesion

```
void a2c(File data, char choice) {
 switch (choice)
   case `a': ... read data ...; break;
   case `b': ... write data to disk ...; break;
   case `c': ... generate data reports; ...
   }
```

Component Coupling

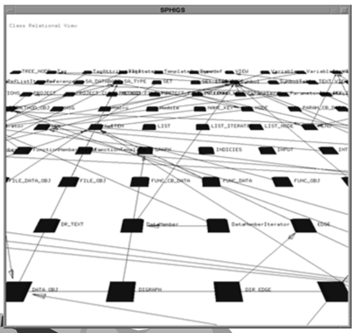
How strong are the interconnections between components?

- Loose (or weak) coupling: a change to one component is not likely to affect other components.
- Shared variables or control information exchange leads to tight (or strong) coupling.

Aim for weak coupling.

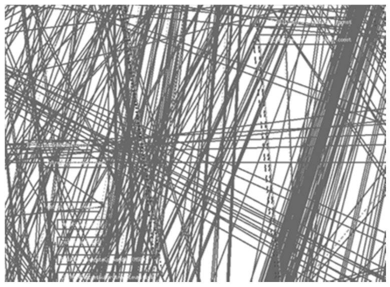
Coupling in the CIV System

- 60 Classes
- 10 KLOC
- 150 Classes with 3rd party classes.



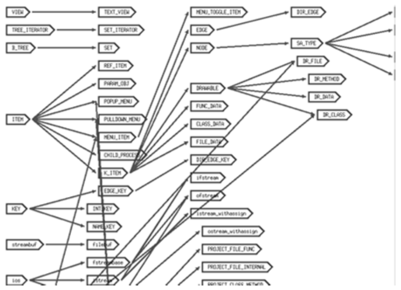
Detailed Call-Graph of CIV

Shows 3% of the entire call graph.

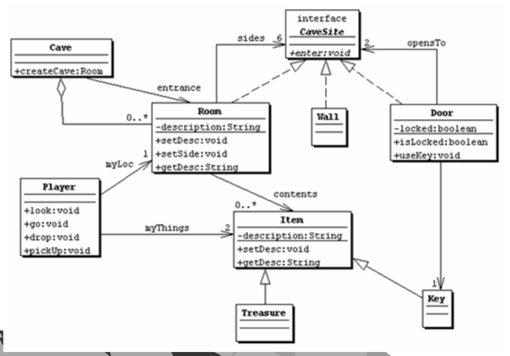


CIV Class Inheritance Graph

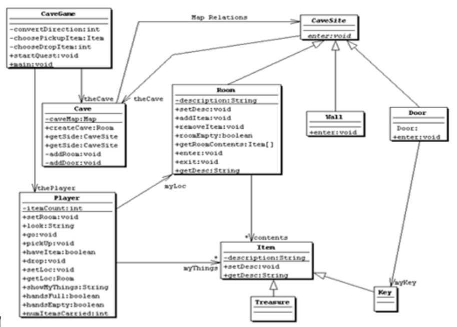
Shows 20% of the entire graph.



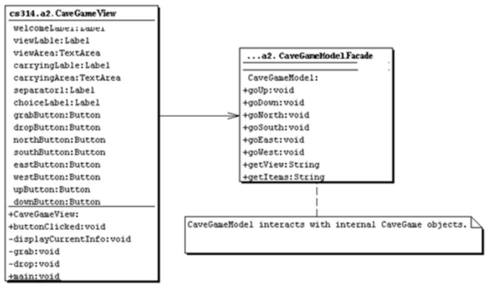
AdventureGame Design-level Class Model



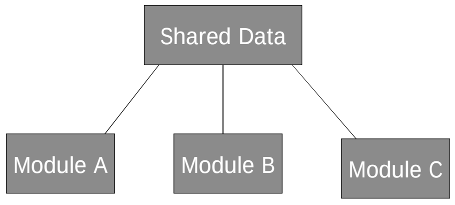
Revised AdventureGame Design



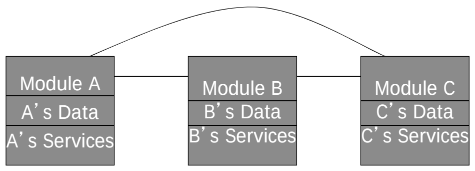
An Architectural View of the AdventureGame Design



Tight Coupling Due to Shared Data



Loose Coupling Through Message Passing

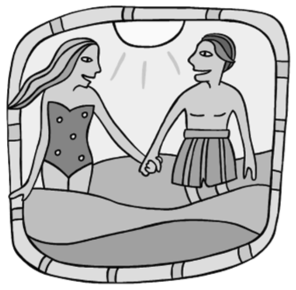


- No shared data.
- Classes are also coupled to their super-classes.

Coupling

- Aim for weak or loose coupling.
- Aim for less coupling.

Minimize the number of couplings and avoid promiscuity!



Coupling Levels: From Weak to Strong Coupling

1. No coupling: modules have no links.
2. Data coupling (low): modules are coupled via simple argument lists.
3. Stamp coupling (low): modules are coupled via a portion of an argument's data structure.

Coupling Levels: From Weak to Strong Coupling

4. Control coupling (moderate): A control flag is passed as a parameter.
5. External coupling (high): modules are coupled to an external environment.

Coupling Levels: From Weak to Strong Coupling

6. Common coupling (high): common references to a global variable.
7. Content coupling (highest): module uses information inside another module, or branches into the middle of a module.

Increasing Cohesion & Reducing Coupling

1. Evaluate the first iteration of a program.
   Explode (split) or implode (merge) modules to improve cohesion or coupling.
2. Minimize structures with high fan-out; Strive for fan-in.
   - Fan-out: the other modules affected by a module.
   - Fan-in: the modules that affect a module.
3. Keep the scope of module effect within the scope of the control of that module.

Increasing Cohesion & Reducing Coupling

4. Evaluate module interfaces to reduce complexity, redundancy, and improve consistency.
5. Define modules whose function is predictable, but avoid modules that are overly restrictive.
6. Strive for controlled entry modules.

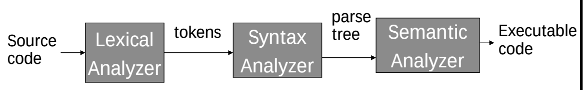
Seek Module Understandability

- Can a module be understood on its own?
- Good designs:
  – Use meaningful names; names should reflect the *intent* of a construct, not its implementation.
  – Have accurate design documentation.
  – Avoid complexity.
- A module must be understood in its design context.

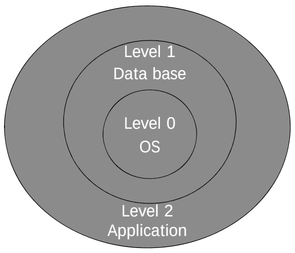
Architectural Configurations

- Hierarchical (tree) structures are common: inheritance & composition.
- Pipeline architectures: compilers.
- Layered: operating systems.
- Client servers.

Pipeline Architecture Example: Compiler Design



Layered or Abstract-Machine Architecture



Distributed Architectures: Client/Server Systems

- Client: service user.
- Server: service provider.
- Clients & Servers can reside on different computers.
- Key concern: the interface connecting clients and servers.

Linking C/S Components

- Pipes: used in UNIX/Linus systems.
- Remote procedure/method calls: invoke program on a different machine.
  rmi in Java
- Client/server SQL interaction: for database queries.

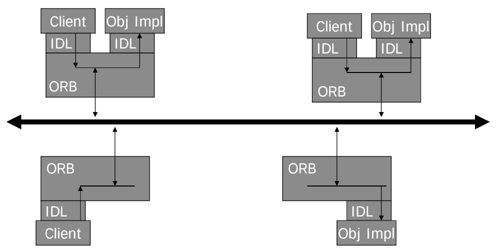
Client/Server Components

- GUI: usually runs on the client.
- Application: may run on either client or server.
- Database management: usually runs on server.
- Middleware: connects client & servers.
  - Network OS support.
  - Object-request brokers.
  - Communication management.

Object-Request Broker (ORB)

- Client object sends message to encapsulated server object.
- An ORB intercepts the message & manages the communication.
  - Finds the server object.
  - Invokes server object methods.
  - Passes data between client & server.

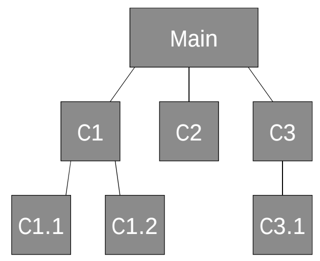
CORBA Architecture



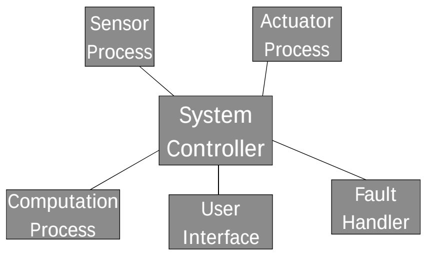
Control Options

- Centralized control:
  - Call-return model uses a hierarchy of components.
  - Manager model for concurrent systems. Controller that spawns tasks.
- Broadcast event-driven control:
  - Subsystems register interest in events.
  - Event handler notifies registered entities when an event occurs.

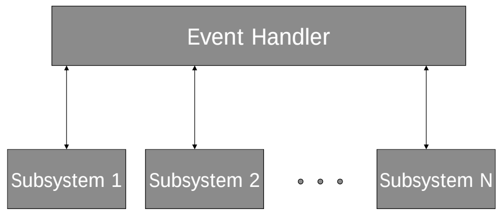
Centralized Control: Call - Return Model



Centralized Control: Manager Model for Concurrent Systems



Broadcast Model for Event - Driven Control



Interrupt-Driven Mechanism

- Interrupt handler for each type of interrupt.
- Each interrupt type is associated to a specific memory location.
- A hardware switch transfers control to a handler.
- Switching is very fast.
- Very difficult to test & validate.

Object-Oriented Design Patterns

- Design pattern:
  "Each pattern describes a problem which occurs over and over again … and then describes the core of the solution" [Christopher Alexander].
- Design patterns focus on flexible designs --- designs that can be more easily adapted and reused.

Design Patterns

- We learn from experience.
- Design pattern purpose: codify good solutions to common design problems.
- Allows us to take advantage of the experiences of other software engineers.

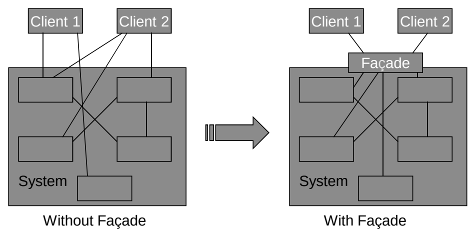
What is a Design Pattern?

- Idioms for structuring object-oriented designs.
- Pattern structure:
  1. Pattern name.
  2. Problem solved by the pattern.
  3. Solution.
  4. Consequences.

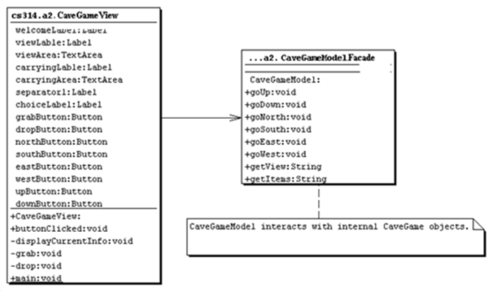
Façade Design Pattern

- Provides a unified interface to a set of interfaces in a subsystem.
- Defines a higher level interface for use by external clients.
- Minimizes dependencies between subsystems.
- A Façade object, rather than internal subsystem objects, becomes the client of all external clients.
- Clients need not know or depend upon details of the subsystems design.

Façade Example



Façade in the AdventureGame Design



Model-View-Controller (MVC) Pattern

- A computer should look like an appliance:
  A TV, radio, calculator, microwave, etc.
- Appliances have:
  - A *view:* TV screen, radio dial, etc.
  - *Controls:* switches, buttons dials.
  - Functional parts: the *model.*

MVC Pattern For GUI Software

- User interfaces are commonly modified.
  - New functionality requires new & updated menus.
  - User interfaces are often adapted for specific customers.
  - User interfaces are often ported to new platforms.
- User interfaces should not be tightly coupled to the functional code.

MVC Components

- Model: the core functionality of the system.
  - This functionality is independent of the output representation or the input behavior.
  - Registers views and controllers.
  - Notifies them when the model state changes.

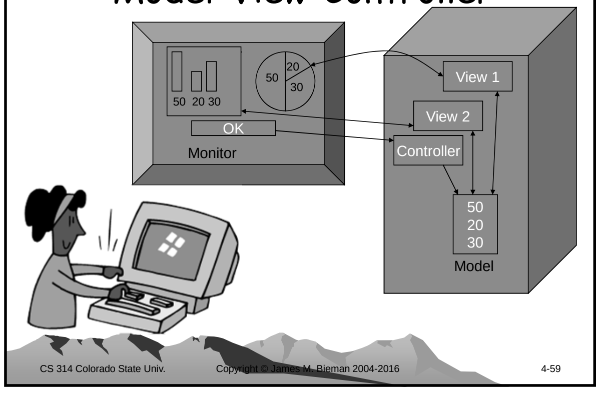
MVC Components

- View: components that display information.
  - Multiple views are allowed.
  - View objects register with the model.
  - They retrieve data from the model when notified of an update.

MVC Components

- Controller: the input component.
  - There may be a controller for each view.
  - The controller receives input.
    - It interprets mouse movement, buttons, and/or keyboard input.
  - Inputs are received as events, which are translated by the controller into requests for service.
  - Service requests are sent to the model or view.
  - Users interact solely via controllers.

Model-View-Controller

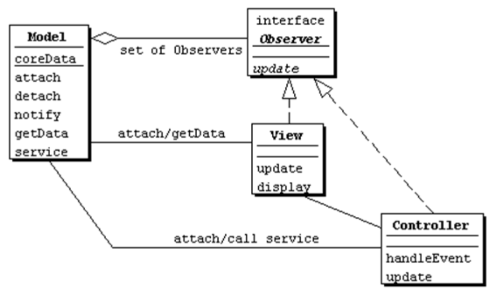


Views Are Easily Changed

- Change a menu into a GUI.
- Add a new display device.
- Change appearance of screens.

No need to change the main functionality (the model).

MVC Class Model



MVC Scenario 1

User input changes the model, views, & controller.

1. Controller accepts user input & requests services of the model.
2. Model performs the service changing model data.
3. Model notifies all registered views & controllers.
4. Views request changed data from model & redisplays changed data & updates menus.
5. Control return to original controller.

MVC Scenario 2

MVC initialization:

1. Model instance is created & its data.
2. For each view:
   1. View object is created; It has a reference to the model.
   2. View registers with the model.
   3. View creates its controller, passing a reference to the model & itself.
   4. Controller registers with the model.
3. The application begins to respond to events.

Simplified MVC

- Full MVC may not be needed.
- Key: separate the MVC functionalities.
- Simple MVC for AdventureGame.java:
  - View object wrapper with a View.displayText() method.
  - Add View attributes to Classes that need it, which are set when initializing the model.
  - CaveGame object acts as the controller.

Abstract Factory Pattern

- Allows us to construct an object configuration, without specifying the components.
- Provides an interface for creating families of related objects without specifying their concrete classes.
- Supports:
  - Systems that should be independent of how its products are created, composed, and represented.
  - Configuring a system with one of multiple families of products.

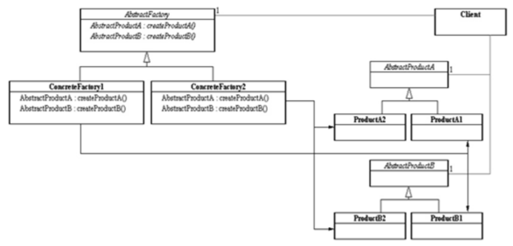
Abstract Factory Implementation Mechanisms

- The method that constructs the configuration has a factory object as a parameter.
- The factory object actually instantiates the objects.
- Client knows only about abstract classes (or interface).

Analogy

- We need to build a car, but don't know what kind of car you are building.
- Cars are built the same way: all have engines, wheels, seats, brakes, etc.
- The specific parts differ between models.
- We specify the construction, but not the specific parts using Abstract Factory.

Abstract Factory Structure


AbstractFactory
AbstractProductA : createProductA()
AbstractProductB : createProductB()
1
Client
ConcreteFactory1
AbstractProductA : createProductA()
AbstractProductB : createProductB()
ConcreteFactory2
AbstractProductA : createProductA()
AbstractProductB : createProductB()
AbstractProductA
1
ProductA2
ProductA1
AbstractProductB
1
ProductB2
ProductB1

Example For Adventure Cave Object Configuration

- Cave configuration may remain the same, but we may have different kinds of rooms, walls, etc. for different levels.
- We describe the cave labyrinth in terms of abstract entities or interfaces.
- We define a different concrete factory for each level.

Possible Code Structure

```
interface CaveFactory {
  Door makeDoor(CaveSite in, Key k, CaveSite
    out);
  Wall makeWall();
  Room makeRoom();
  Player makePlayer();
  Key makeKey();
}
```

Building a Cave Configuration

```
public class Labyrinth{
  …
  public Room createLabyrinth(CaveFactory f) {
     Room outside = f.makeRoom();
     Room cell = f.makeRoom();
     Key k = f.makeKey();
     Door d = f.makeDoor(outside, k, cell);
     ….
     return cell;
}
```

Use a different CaveFactory for each level

Level 1 Concrete Cave Factory

```
Class Level1CaveFactory
    implements CaveFactory {
  // Builds Level 1 rooms, walls, doors, …
  Door makeDoor(CaveSite in, Key k, CaveSite out){
      …}
  Wall makeWall() {…}
  Room makeRoom() {…}
  Player makePlayer() {…}
  Key makeKey() {…}
}
```

Level 2 Concrete Cave Factory

```
Class Level2CaveFactory
   implements CaveFactory {
  // Builds Level 2 rooms, walls, doors, …
  Door makeDoor(CaveSite in, Key k, CaveSite out){
      …}
  Wall makeWall() {…}
  Room makeRoom() {…}
  Player makePlayer() {…}
  Key makeKey() {…}
}
```

With Abstract Factory Pattern

- You don't have to modify code in the basic cave system structure to change levels.
- A Concrete Factory can extend another Concrete Factory.
  A Level n+1 factory can extend a Level n factory.

Broker Pattern for Client/Server Architectures

- Decouples clients and servers.
  - Servers register with the brokers.
  - Clients send brokers requests for services.
  - Brokers locate an appropriate server, forwards request to server, and transmits results back to the client.
  - Clients don't deal with low-level communication details.
- Allows dynamic change, addition, deletion, & relocation of objects.

Broker Components

- Clients: access servers.
  - Example: WWW browsers.
  - Need not know the location of servers.
- Servers: implements objects.
  - Export interfaces: interface definition language (IDL) or a binary standard.
  - Example: WWW servers give access to HTML pages, CGI scripts, applets.

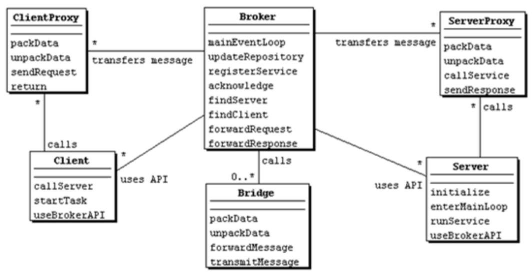
Broker Components

- Brokers:
  - Transmits requests from client to server & results from server to client.
  - Must have a way to locate & activate servers.
  - Offers:
    - APIs to clients & servers.
    - operations to register servers.
    - Operations to invoke server methods.
  - May pass a request to a remote broker.

Broker Components

- Client-side proxies: layer between clients and the broker.
  - Allows a remote object to appear to be local.
  - Hides implementation details from client.
- Server-side proxies: layer between servers & broker.
  Unpack incoming messages.
- Bridges: hide implementation details when 2 brokers communicate.

Broker Pattern Structure



Broker Scenario 1

Server registers with the local broker:

1. Broker is initialized, then enters event loop.
2. User starts a server application. It registers with the broker.
3. The broker receives the registration request, records & acknowledges it.
4. Server waits for incoming client requests.

Broker Scenario 2

Client sends a request to a local server:

1. Client invokes a remote server object method.
2. Client-side proxy packages parameters into a message and sends it to the local broker.
3. The broker finds location of the server & sends the message to the server-side proxy.
4. Server-side proxy unpacks parameters etc and invokes the requested service.

Broker Scenario 2

5. Server returns result to server-side proxy, which packages it into a message & sends it to the broker.
6. The broker forwards the response to the client-side proxy.
7. The client-side proxy unpacks the results & returns it to the client.

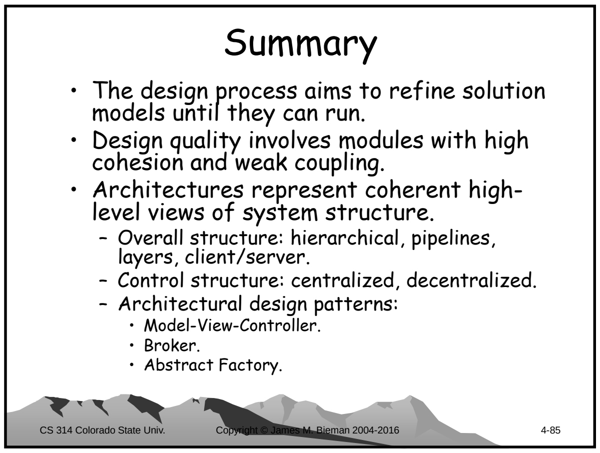
Broker Scenario 3

Interaction between remote brokers using bridges:

1. Broker A receives an incoming request. It locates the remote server & forwards the request to a remote broker.
2. Bridge Message is passed from Broker A to Bridge A.
3. A converts the message to a common protocol & sends the message to Bridge B at the server site.
4. Bridge B maps the request into a Broker B format.
5. Broker B handles the request.

Broker Pattern Examples

- CORBA: common object request broker architecture, an Object Management Group (OMG) standard, with an Interface Definition Language (IDL).
- IBM SOM/DSOM.
- Microsoft OLE: defines a binary standard for accessing server interfaces.
- The WWW.



Summary

- The design process aims to refine solution models until they can run.
- Design quality involves modules with high cohesion and weak coupling.
- Architectures represent coherent high-level views of system structure.
  - Overall structure: hierarchical, pipelines, layers, client/server.
  - Control structure: centralized, decentralized.
  - Architectural design patterns:
    - Model-View-Controller.
    - Broker.
    - Abstract Factory.

CS 314 Colorado State Univ. Copyright © James M. Bieman 2004-2016 4-85