CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Lecture 8: Modularity

Gregory Gay

TDA 594/DIT 593 - November 25, 2021

Annotation-Based Representation

- Code in common code base.
- Code related to a feature is marked.
  - Preprocessor annotations, if-statements.
- Code belonging to deselected features:
  - ignored (load-time, run-time)
  - removed (compile-time).

Composition-based Representation

- Feature code in dedicated location.
  - Class, file, package, service
- Selected units combined to form product.
- Requires clear mapping between features and units

Today’s Goals

- Frameworks
  - Libraries of extendable base implementations.
  - Subclass a template class (white box), implement objects following an interface and register them (black box).
- Components/Services
  - Standalone units with explicit interfaces.
  - Can be reused in other systems.
  - Form a system as part of a broader architecture.

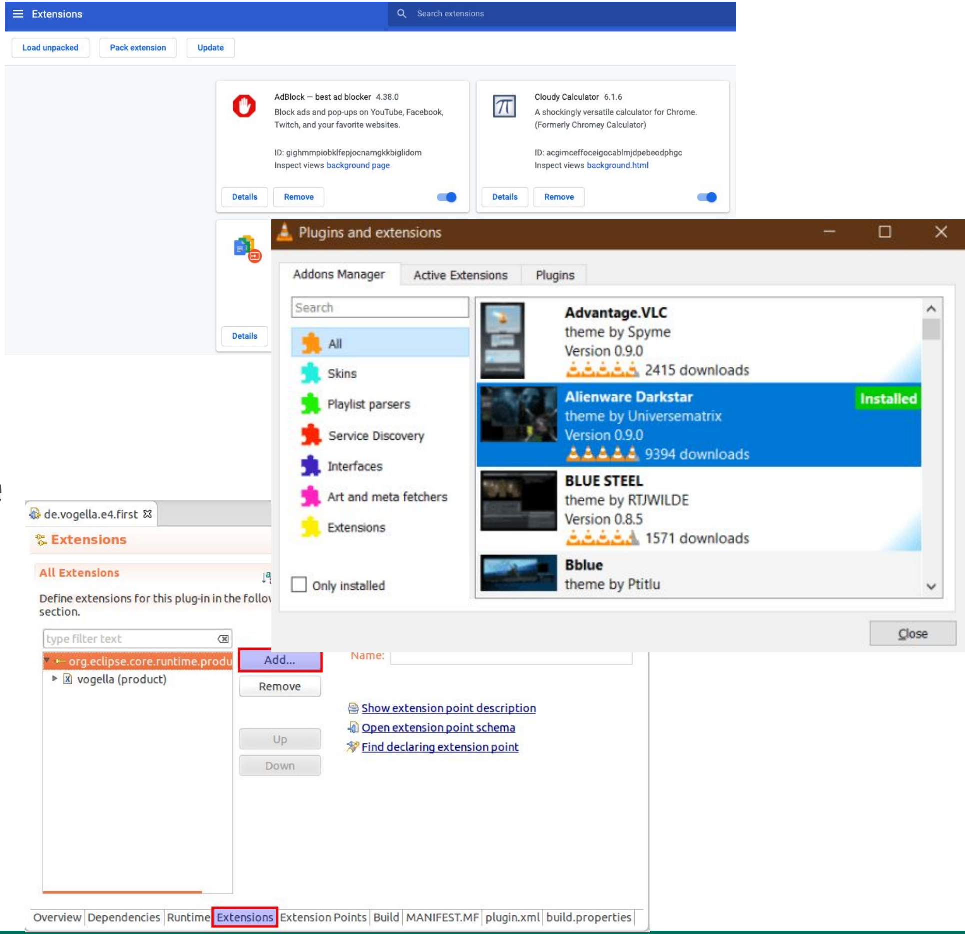
Frameworks and Libraries

Frameworks

- A collection of classes that represent solutions to related problems.
  - Base implementation that can be extended with new custom use cases.
  - Provides extension points (“**hot spots**”)
- Framework is responsible for main control flow, asks extensions for custom behavior.

Frameworks

- Used in web browsers, graphics editing, media players, IDEs.
- In product line, a feature developed as a plug-in.
  - Select plug-ins based on feature selection.



AdBlock — best ad blocker 4.38.0
Block ads and pop-ups on YouTube, Facebook, Twitch, and your favorite websites.
ID: gighmmpiobklfepjocnamgkkbiglidom
Inspect views background page

Cloudy Calculator 6.1.6
A shockingly versatile calculator for Chrome. (Formerly Chromey Calculator)
ID: acgimceffoceigocablmjdpebeodphgc
Inspect views background.html

Advantage.VLC
theme by Spyme
Version 0.9.0
2415 downloads

Alienware Darkstar
theme by Universematrix
Version 0.9.0
9394 downloads

BLUE STEEL
theme by RTJWILDE
Version 0.8.5
1571 downloads

Bblue
theme by Ptitlu

Define extensions for this plug-in in the follo section.

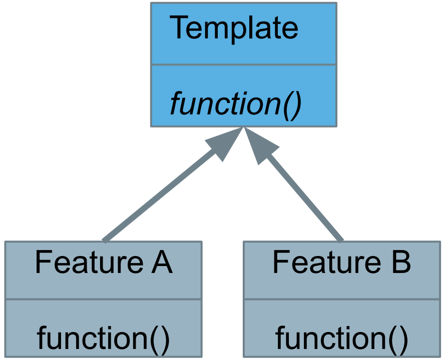
org.eclipse.core.runtime.produ
vogella (product)

White-Box Frameworks

- Abstract class with concrete subclasses.
  - Defines default behaviors (template methods).
  - Extensions implemented as new concrete classes that override these methods.
- Directly implemented in existing codebase.
  - Requires access to source code.
  - Free to interact with existing code, access base implementation.

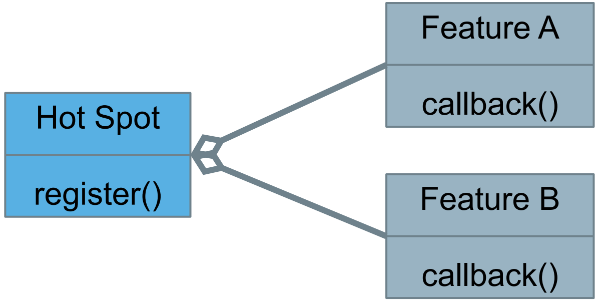
White-Box Frameworks

- Overriding existing behavior allows flexibility.
  - … But requires detailed understanding of low-level implementation.
  - Fails to protect existing code from extensions.
- Often used for libraries
  - GUI elements, data structures
- Features implemented as subclasses.
  - Best for alternative features (choose-one).



Black-Box Frameworks

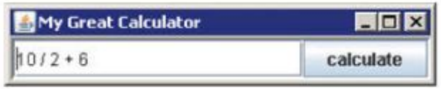
- Separate code and extensions through interfaces.
  - Each feature is a separate plug-in.
  - Plug-ins registered at hot spots.
    - E.g., Observers, strategies.
  - One or more plug-ins can be attached.



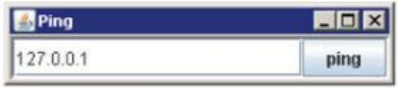
Black-Box Frameworks

- Developers only need to understand interfaces.
  - Easier to understand framework.
  - Internal functions, information protected.
  - Can only extend designated hot-spots.
- Limits flexibility, but decouples framework/extensions.
  - Can independently develop/distribute extensions.

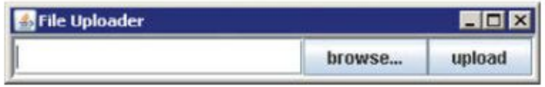
Implementation Example

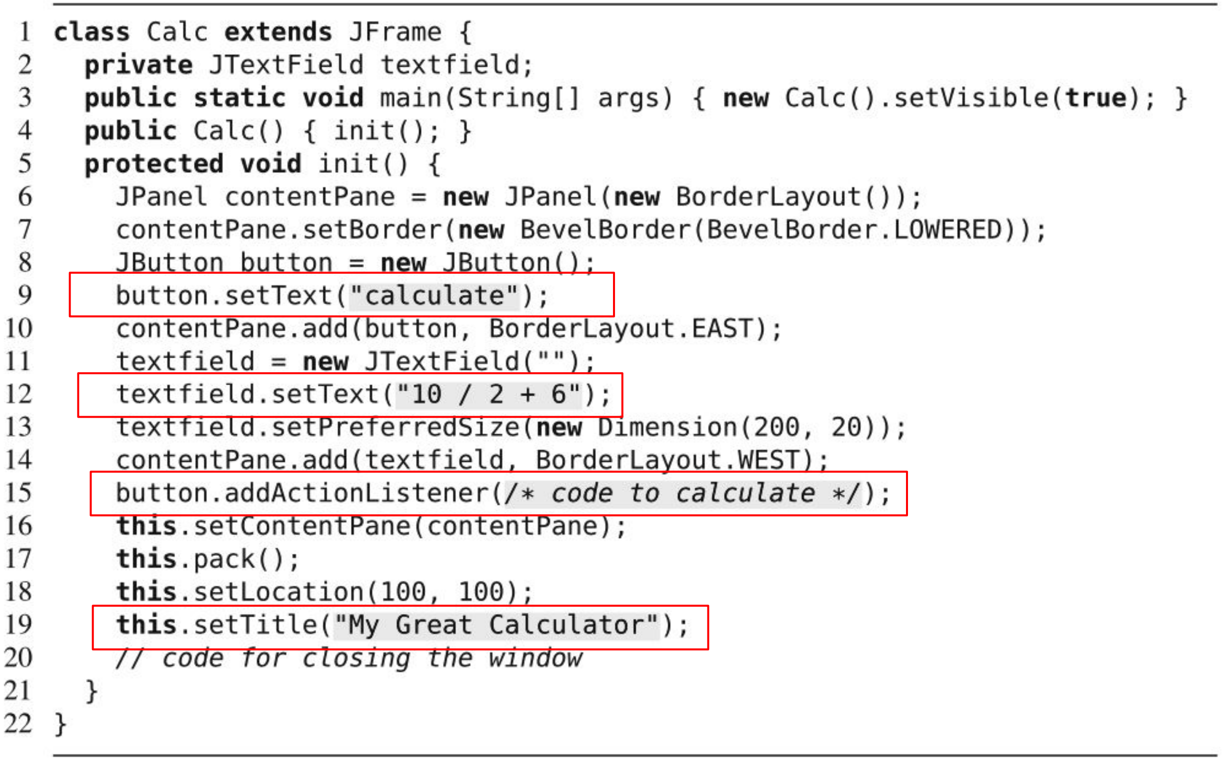


10 / 2 + 6



127.0.0.1

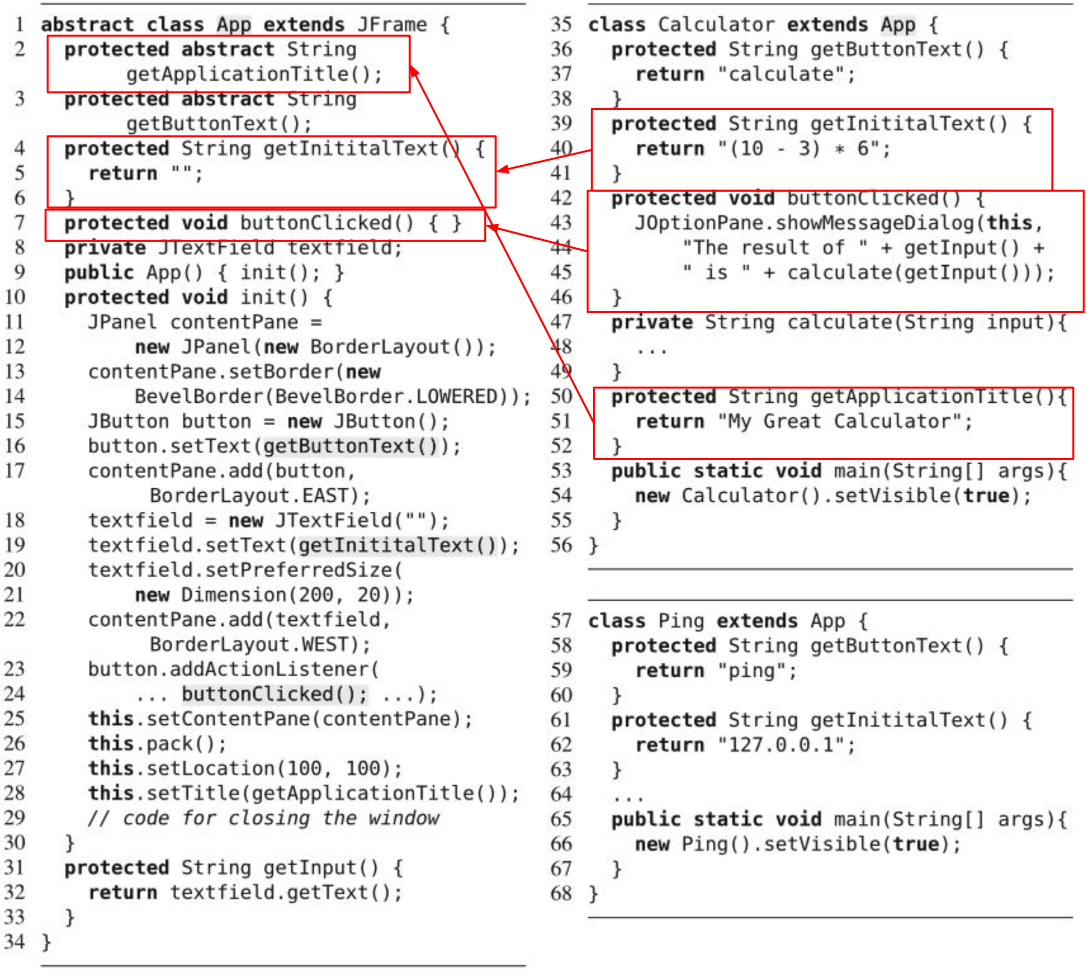




```
class Calc extends JFrame {
  private JTextField textfield;
  public static void main(String[] args) { new Calc().setVisible(true); }
  public Calc() { init(); }
  protected void init() {
    JPanel contentPane = new JPanel(new BorderLayout());
    contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
    JButton button = new JButton();
    button.setText("calculate");
    contentPane.add(button, BorderLayout.EAST);
    textfield = new JTextField("");
    textfield.setText("10 / 2 + 6");
    textfield.setPreferredSize(new Dimension(200, 20));
    contentPane.add(textfield, BorderLayout.WEST);
    button.addActionListener(/* code to calculate */);
    this.setContentPane(contentPane);
    this.pack();
    this.setLocation(100, 100);
    this.setTitle("My Great Calculator");
    // code for closing the window
  }
}
```

White-Box

- Abstract class implements base behavior.
  - Defines abstract or default methods that will be extended.
  - Subclasses override those methods.



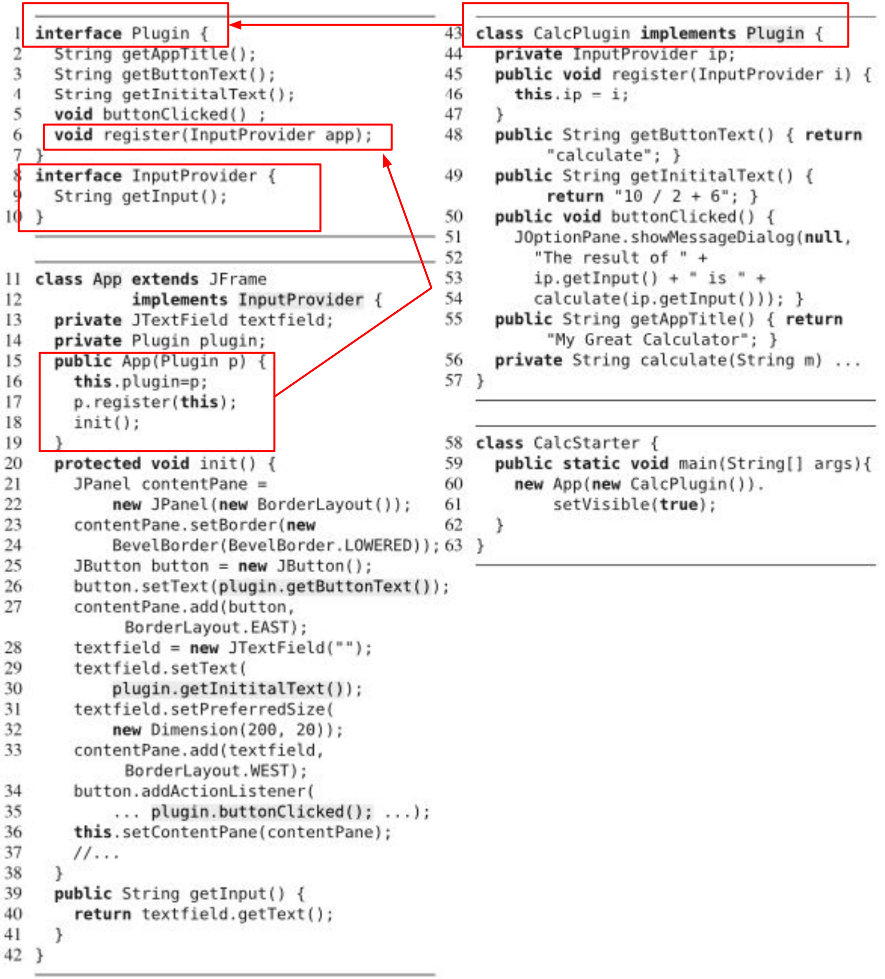
```
abstract class App extends JFrame {
  protected abstract String
      getApplicationTitle();
  protected abstract String
      getButtonText();
  protected String getInititalText() {
    return "";
  }
  protected void buttonClicked() { }
  private JTextField textfield;
  public App() { init(); }
  protected void init() {
    JPanel contentPane =
        new JPanel(new BorderLayout());
    contentPane.setBorder(new
        BevelBorder(BevelBorder.LOWERED));
    JButton button = new JButton();
    button.setText(getButtonText());
    contentPane.add(button,
         BorderLayout.EAST);
    textfield = new JTextField("");
    textfield.setText(getInititalText());
    textfield.setPreferredSize(
        new Dimension(200, 20));
    contentPane.add(textfield,
         BorderLayout.WEST);
    button.addActionListener(
        ... buttonClicked(); ...);
    this.setContentPane(contentPane);
    this.pack();
    this.setLocation(100, 100);
    this.setTitle(getApplicationTitle());
    // code for closing the window
  }
  protected String getInput() {
    return textfield.getText();
  }
}
```

```
class Calculator extends App {
  protected String getButtonText() {
    return "calculate";
  }
  protected String getInititalText() {
    return "(10 - 3) * 6";
  }
  protected void buttonClicked() {
    JOptionPane.showMessageDialog(this,
      "The result of " + getInput() +
      " is " + calculate(getInput()));
  }
  private String calculate(String input){
    ...
  }
  protected String getApplicationTitle(){
    return "My Great Calculator";
  }
  public static void main(String[] args){
    new Calculator().setVisible(true);
  }
}
```

```
class Ping extends App {
  protected String getButtonText() {
    return "ping";
  }
  protected String getInititalText() {
    return "127.0.0.1";
  }
  ...
  public static void main(String[] args){
    new Ping().setVisible(true);
  }
}
```

Black-Box

- Extensions implement a defined interface.
  - Register with the framework to provide needed functionality.
  - Can also use interface to surface information from framework in app (`InputProvider`)



```
interface Plugin {
  String getAppTitle();
  String getButtonText();
  String getInititalText();
  void buttonClicked() ;
  void register(InputProvider app);
}
interface InputProvider {
  String getInput();
}
```

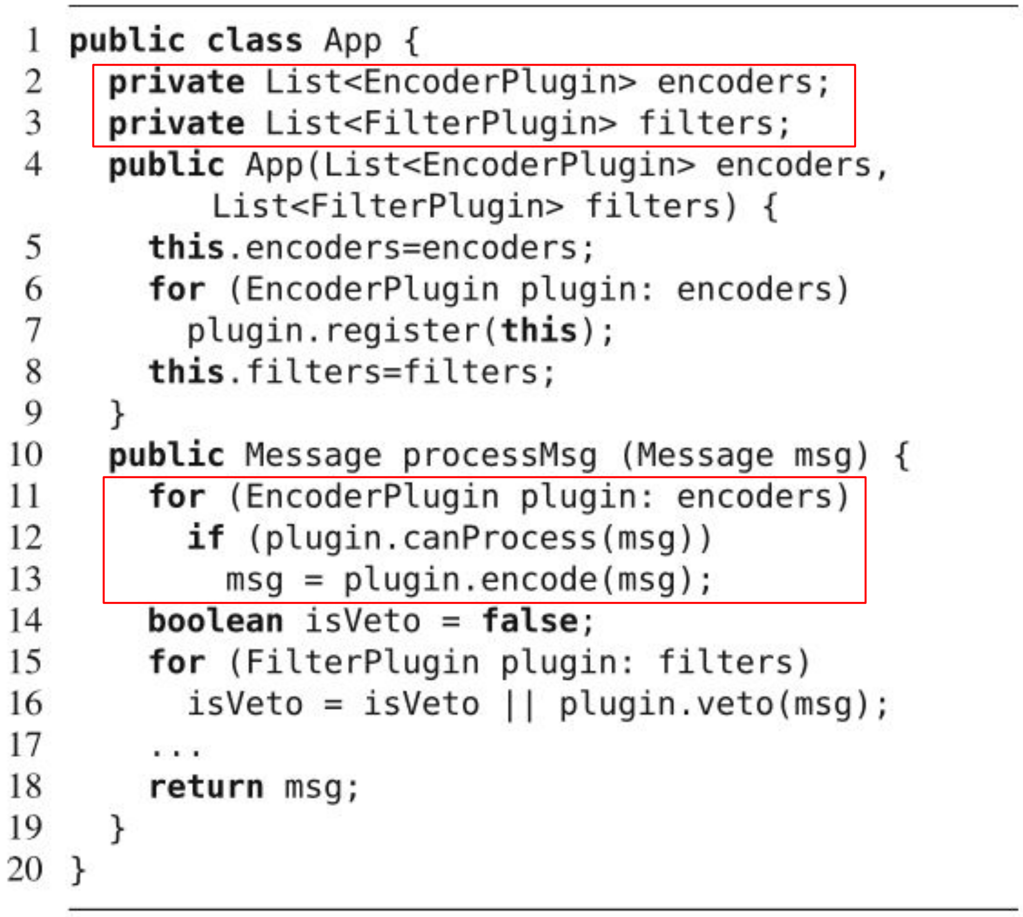
```
class App extends JFrame
          implements InputProvider {
  private JTextField textfield;
  private Plugin plugin;
  public App(Plugin p) {
    this.plugin=p;
    p.register(this);
    init();
  }
  protected void init() {
    JPanel contentPane =
        new JPanel(new BorderLayout());
    contentPane.setBorder(new
        BevelBorder(BevelBorder.LOWERED));
    JButton button = new JButton();
    button.setText(plugin.getButtonText());
    contentPane.add(button,
         BorderLayout.EAST);
    textfield = new JTextField("");
    textfield.setText(
        plugin.getInititalText());
    textfield.setPreferredSize(
        new Dimension(200, 20));
    contentPane.add(textfield,
         BorderLayout.WEST);
    button.addActionListener(
        ... plugin.buttonClicked(); ...);
    this.setContentPane(contentPane);
    //...
  }
  public String getInput() {
    return textfield.getText();
  }
}
```

```
class CalcPlugin implements Plugin {
  private InputProvider ip;
  public void register(InputProvider i) {
    this.ip = i;
  }
  public String getButtonText() { return
       "calculate"; }
  public String getInititalText() {
       return "10 / 2 + 6"; }
  public void buttonClicked() {
    JOptionPane.showMessageDialog(null,
      "The result of " +
      ip.getInput() + " is " +
      calculate(ip.getInput())); }
  public String getAppTitle() { return
      "My Great Calculator"; }
  private String calculate(String m) ...
}
```

```
class CalcStarter {
  public static void main(String[] args){
    new App(new CalcPlugin()).
        setVisible(true);
  }
}
```

Black-Box

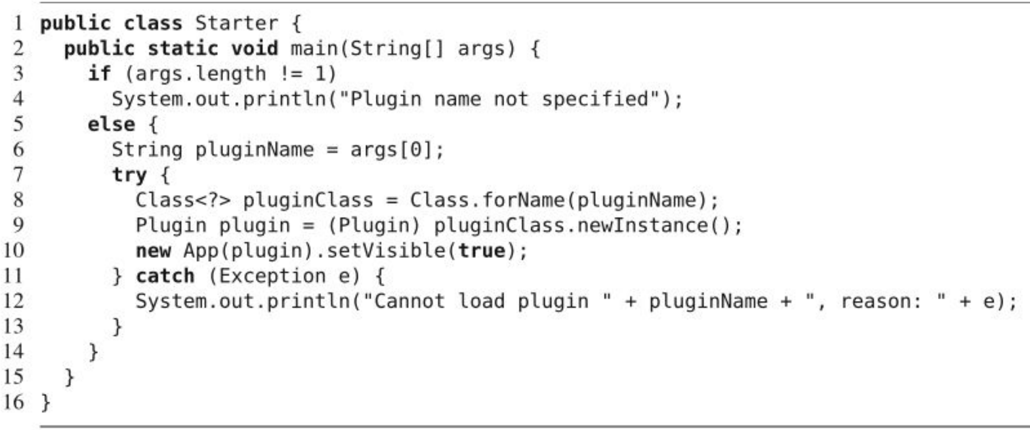
- Can register multiple extensions in a list.
- Can extend with multiple types of extensions at same point.



```
public class App {
  private List<EncoderPlugin> encoders;
  private List<FilterPlugin> filters;
  public App(List<EncoderPlugin> encoders,
       List<FilterPlugin> filters) {
    this.encoders=encoders;
    for (EncoderPlugin plugin: encoders)
      plugin.register(this);
    this.filters=filters;
  }
  public Message processMsg (Message msg) {
    for (EncoderPlugin plugin: encoders)
      if (plugin.canProcess(msg))
        msg = plugin.encode(msg);
    boolean isVeto = false;
    for (FilterPlugin plugin: filters)
      isVeto = isVeto || plugin.veto(msg);
    ...
    return msg;
  }
}
```

Loading Plug-Ins

- Often loaded when application is executed.
  - Command-line parameter, config file, directory.
- Sets up framework with detected plug-ins.



```
public class Starter {
  public static void main(String[] args) {
    if (args.length != 1)
      System.out.println("Plugin name not specified");
    else {
      String pluginName = args[0];
      try {
        Class<?> pluginClass = Class.forName(pluginName);
        Plugin plugin = (Plugin) pluginClass.newInstance();
        new App(plugin).setVisible(true);
      } catch (Exception e) {
        System.out.println("Cannot load plugin " + pluginName + ", reason: " + e);
      }
    }
  }
}
```

- Can check whether plug-in implements correct interface, check dependencies, check constraints between plug-ings.
- May use a built-in extension manager (Chrome)

Discussion

- Composition-based, often load-time, approach.
  - **Uniform:** Can be implemented similarly across many languages, technologies.
  - **Traceable:** Direct correspondence from feature to code (one plug-in = one feature)
- Black-box frameworks can encode alternative and optional features easily and systematically.
- White-box can encode alternative features, but harder to blend features.

Discussion

- **Separation of Concerns:**
  - Interfaces encapsulate framework from plug-ins.
  - Plug-ins developed separately from framework, as long as interface is followed.
- **Information Hiding:** Can understand feature by only looking at plug-in code.
- **Modularity:** Independent developers can develop their own extensions.

Discussion

- **Pre-planning Effort:** Must anticipate hot-spots and design interfaces and templates.
    - If needed information not exposed to extensions, framework must be refactored.
    - Interfaces cannot change without changing all plug-ins.
- Changing a framework can be inflexible.

Discussion

- Plug-ins can be reused in versions of the same framework, but not in other frameworks.
  - Tied closely to implementation.
- Introduce development and run-time overhead.
  - Must write additional code.
  - Can lead to over-complex design.
  - More code must be executed, slowing the system.
  - Limit to few well-defined extension points in code.

Components and Services

Components

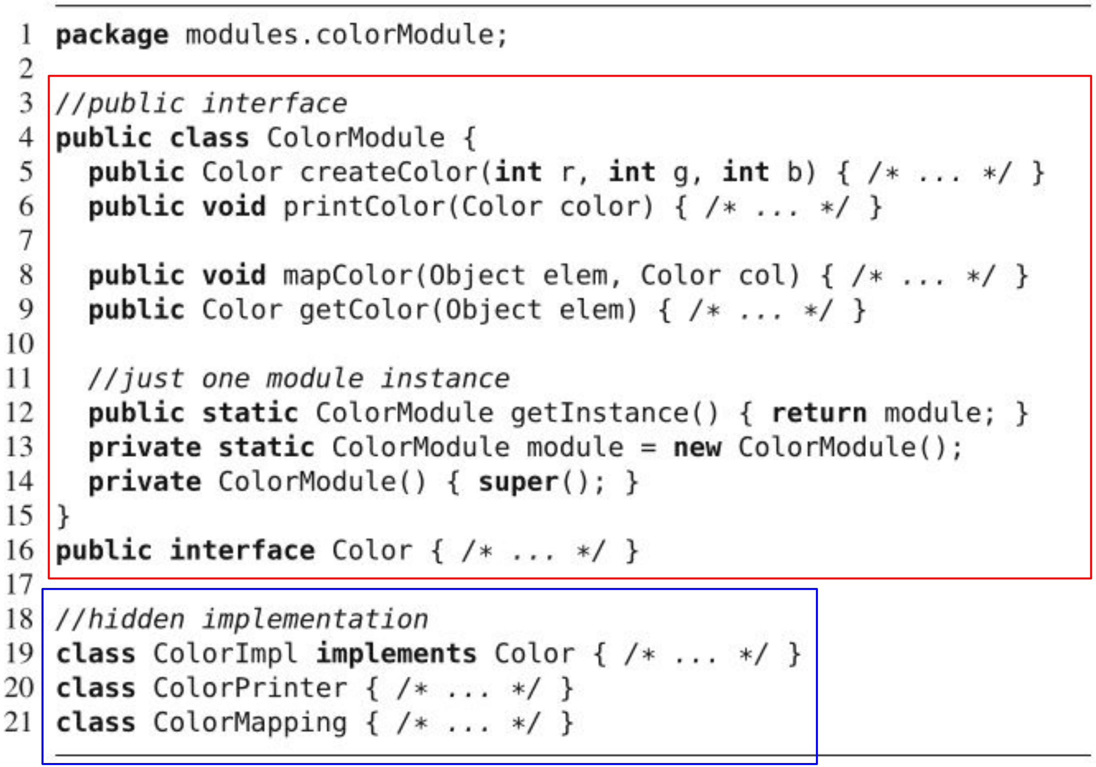
- A **component** is a standalone unit with specified interfaces and explicit dependencies.
  - Can be deployed independently.
  - Can be reused in many systems.
  - Can vary from one class to many.
- Developers can choose to implement their own components or work with existing ones.
  - Requires compatible interfaces and data.

Services

- A form of component focused on standardization, interoperability, and distribution.
  - Reachable over standard protocols.
    - HTTP
  - Can often look up services from a registry.
    - NPM for JavaScript
  - Communication standardized so underlying language does not matter.
    - REST API

Simple Example

- Define public interface (class ColorModule, interface Color)
- Hide implementation (private/package-level visibility)
- Can integrate into code or as JAR file.



```java
package modules.colorModule;

//public interface
public class ColorModule {
  public Color createColor(int r, int g, int b) { /* ... */ }
  public void printColor(Color color) { /* ... */ }

  public void mapColor(Object elem, Color col) { /* ... */ }
  public Color getColor(Object elem) { /* ... */ }

  //just one module instance
  public static ColorModule getInstance() { return module; }
  private static ColorModule module = new ColorModule();
  private ColorModule() { super(); }
}
public interface Color { /* ... */ }

//hidden implementation
class ColorImpl implements Color { /* ... */ }
class ColorPrinter { /* ... */ }
class ColorMapping { /* ... */ }
```

Components vs Plug-Ins

- Both result in encapsulated modules.
  - Enabling traceability, information hiding.
- Difference in **automation potential** and **reuse**.
  - Plug-ins tailored to one framework.
    - Product can be generated by loading only the needed plug-ins.
    - Plug-ins designed for that framework.
    - Hard to reuse.
  - Components can be reused.
    - But require glue code to integrate.

Components vs Plug-Ins

- Components can be encoded in many languages.
- Both allow compile-time product derivation.
- Interfaces for both are difficult to evolve once designed.
  - Others may depend on current interface definition.
- Both add overhead from interfacing/communication.

Sizing Components

- A component can contain a lot of functionality or only offer a single, small function.
  - A complex component is *easier to use* in the project it was developed for, but *hard to reuse* elsewhere.
  - Small components are easy to reuse in many projects, but add communication overhead and glue code.
  - Trade-off between *use* and *reuse.*

Sizing Components

- Non-trivial to size components.
- Domain analysis helps in SPL development.
  - Which functionality will be reused in different products?
  - If functions are *always* used together, package them together as a component.
  - If a function is only used in a subset of products, it can be packages as a separate component.

Let’s take a break!

Composing Components into a Software Architecture

Static Structures

- **Static structures** define system's internal components and their arrangement.
  - Software: services, classes, packages.
  - Data: Database entries/tables, data files.
  - Hardware: Servers, CPUs, disks, networking.
- Static arrangement defines associations, relationships, or connectivity between components.

Static Structure Arrangement

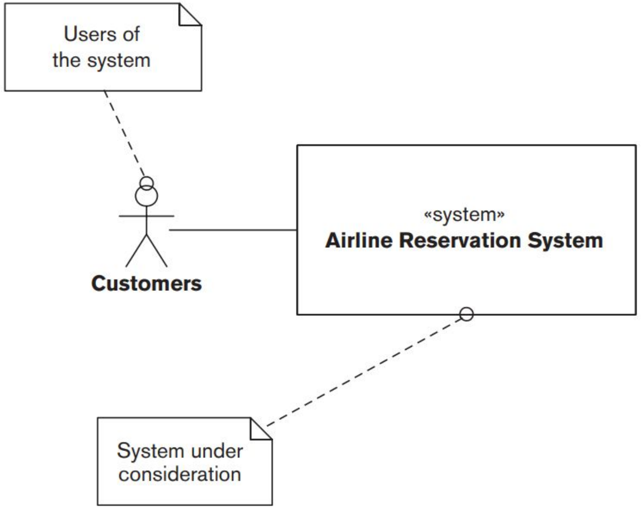
- Software:
  - Relationships define **hierarchy** (inheritance) or **dependency** (use of variables or methods).
- Data:
  - Relationships define how data items are linked.
- Hardware:
  - Relationships define physical interconnections between hardware components.

Dynamic Structures

- **Dynamic structures** define system’s runtime elements and their interactions.
- Flow of information
  - A sends messages to B
- Flow of control
  - A.action() invokes B.action()
- Effect an action has on data.
  - Entry E is created, updated, and destroyed.

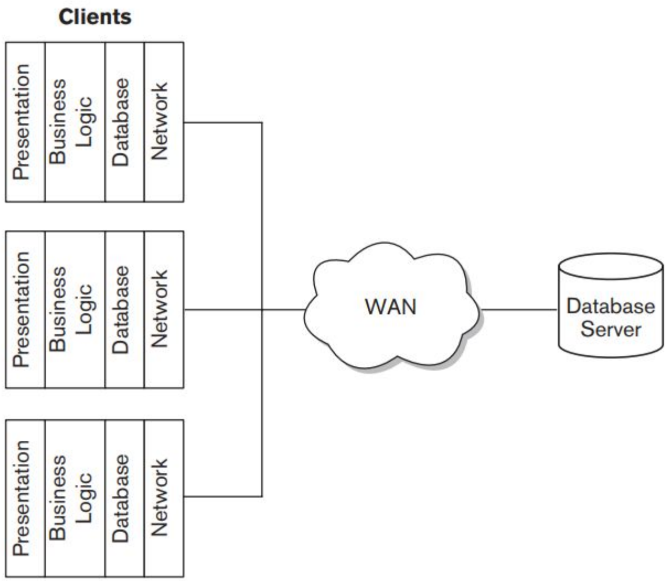
Airline Reservation System

- Allows seat booking, updating, cancellation, upgrading, transferring.
- **Externally visible behavior:**
  - How it responds to submitted transactions.
- **Quality properties of interest:**
  - Average response time, max throughput, availability



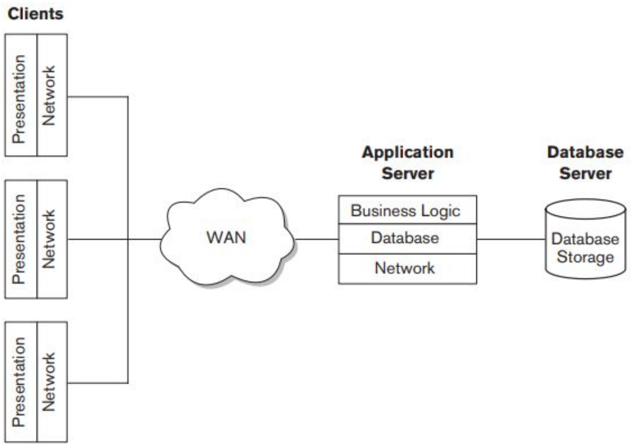
Option 1: Client/Server Architecture

- Clients communicate with a central server (with a database) over a network.
- **Static Structure:** Client programs, broken into layered elements, a server, and connections.
- **Dynamic Structure:** Request/response model.



Option 2: “Thin Client” Architecture

- Clients communicate with a central server (with a database) over a network.
- **Static Structure:** Client only perform presentation. Server performs logic computation.
- **Dynamic Structure:** Request/response model. Requests submitted to application server, then database server.

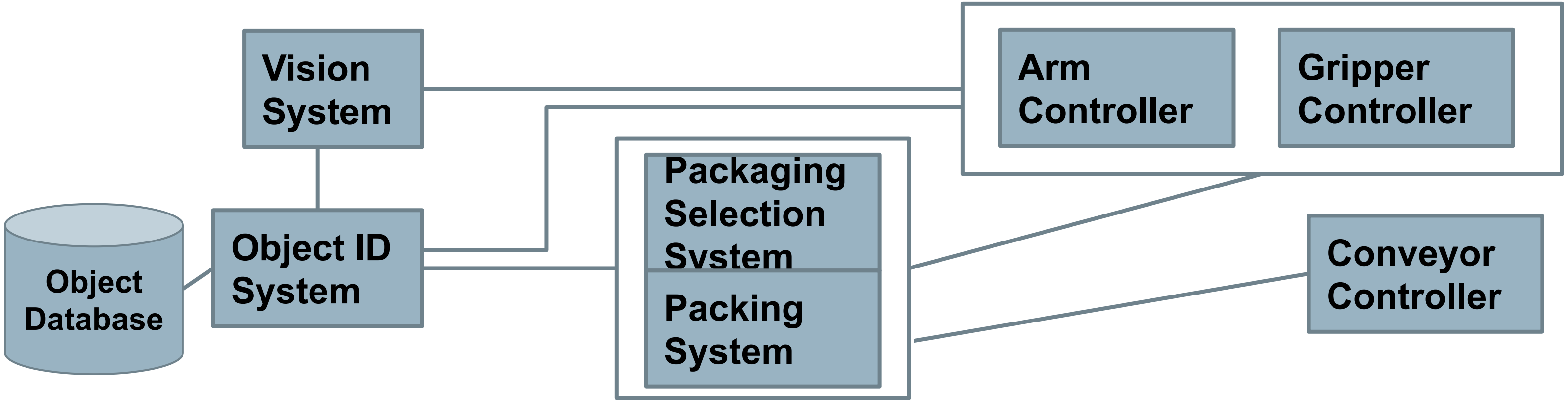


Which Would You Choose?

- Same external behavior, may differ in performance.
  - First is simpler.
  - Second might be more scalable or more secure.
- Must select a candidate architecture that satisfies all requirements and meets quality goals.
- Extent that a architecture exhibits behaviors and performance must be studied further.

Static Structuring

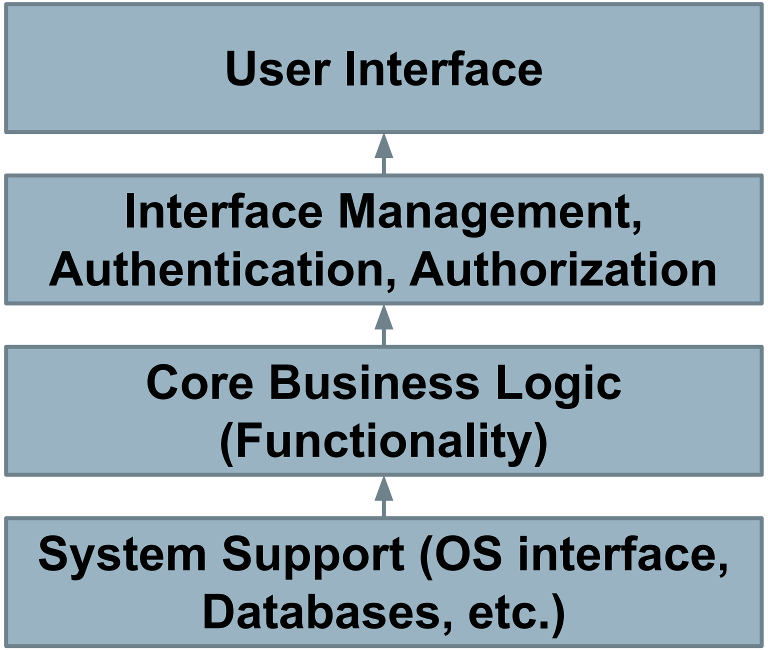
- Decompose the system into components.
- Visualized as structured blocks.



Basic Architectural Styles

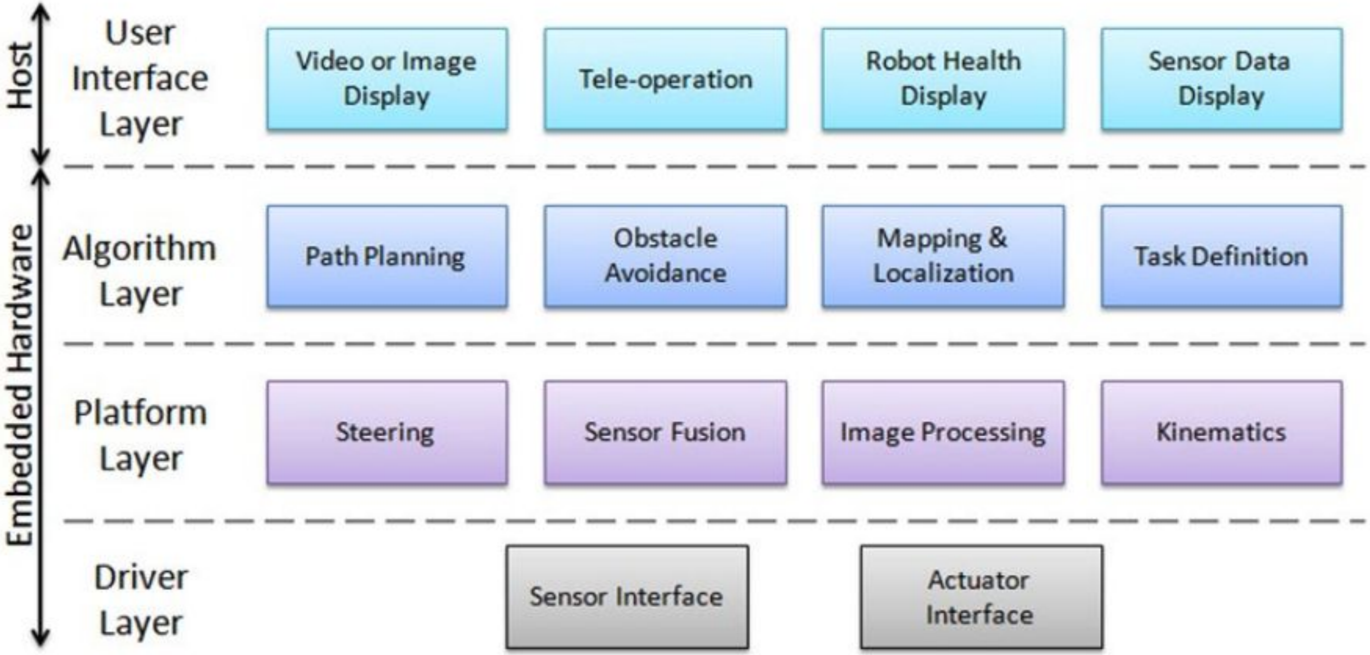
- Common styles: layered, shared repository, client/server, pipe & filter
- The style used affects performance, robustness, maintainability, etc.
- Complex systems might not follow a single model - mix and match for subsystems.

Layered Model


User Interface
Interface Management, Authentication, Authorization
Core Business Logic (Functionality)
System Support (OS interface, Databases, etc.)

- Components organized into layers
  - Each layer only dependent on the previous layer.
  - May be multiple components in a single layer.
- Allows components to change independently.
- Supports incremental development.

Robot Example



Host	User Interface Layer	Video or Image Display	Tele-operation	Robot Health Display	Sensor Data Display
Embedded Hardware	Algorithm Layer	Path Planning	Obstacle Avoidance	Mapping & Localization	Task Definition
	Platform Layer	Steering	Sensor Fusion	Image Processing	Kinematics
	Driver Layer	Sensor Interface	Actuator Interface		

Layered Model Characteristics

Advantages

- Allows replacement of entire layers as long as interface is maintained.
- Changes only impact the adjacent layer.
- Redundant features (authentication) in each layer can enhance security and dependability.

Disadvantages

- Clean separation between layers is difficult.
- Performance can be a problem because of multiple layers of processing between call and return.

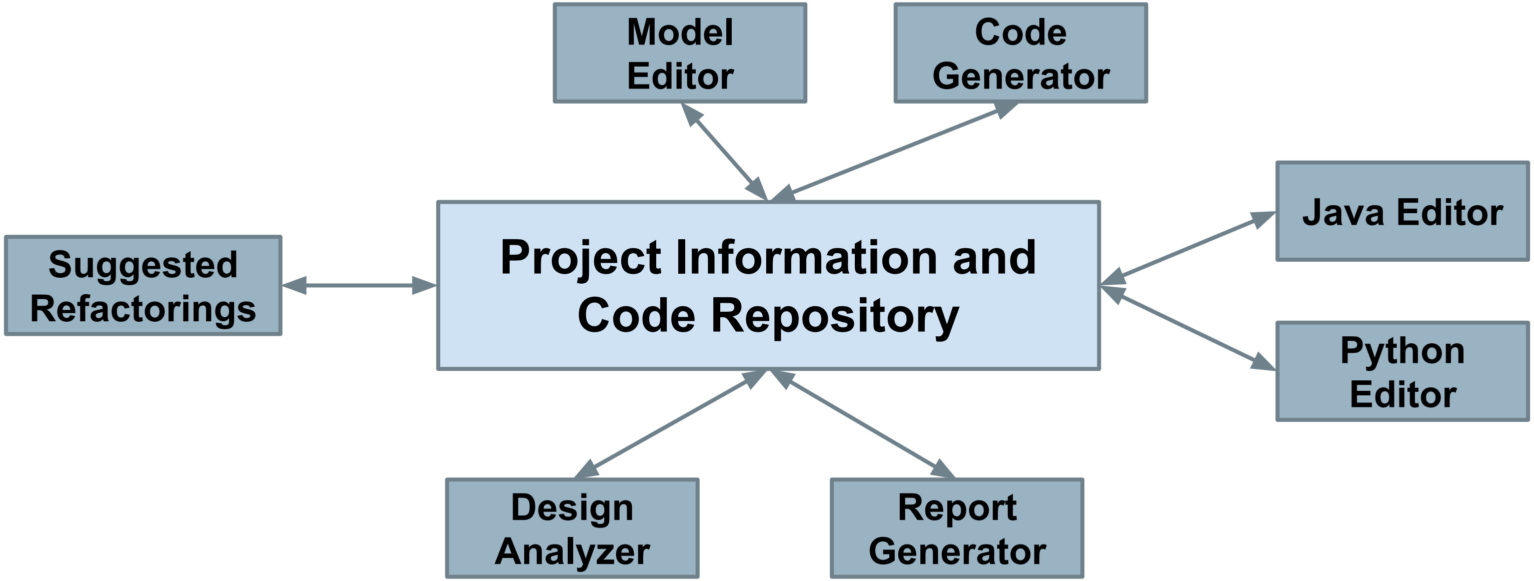
The Repository Model

Components often exchange and work with the same data. This can be done in two ways:

- Each component maintains its own data and passes it to other components.
- **Shared data held in central repository and accessed by all components.**

Repository model is structured around the latter.

IDE Example


Model Editor
Code Generator
Suggested Refactorings
Project Information and Code Repository
Java Editor
Python Editor
Design Analyzer
Report Generator

Repository Model Characteristics

Advantages

- Efficient way to share data.
- Components can be independent.
  - May be more secure.
- All data can be managed consistently (centralized backup, security, etc)

Disadvantages

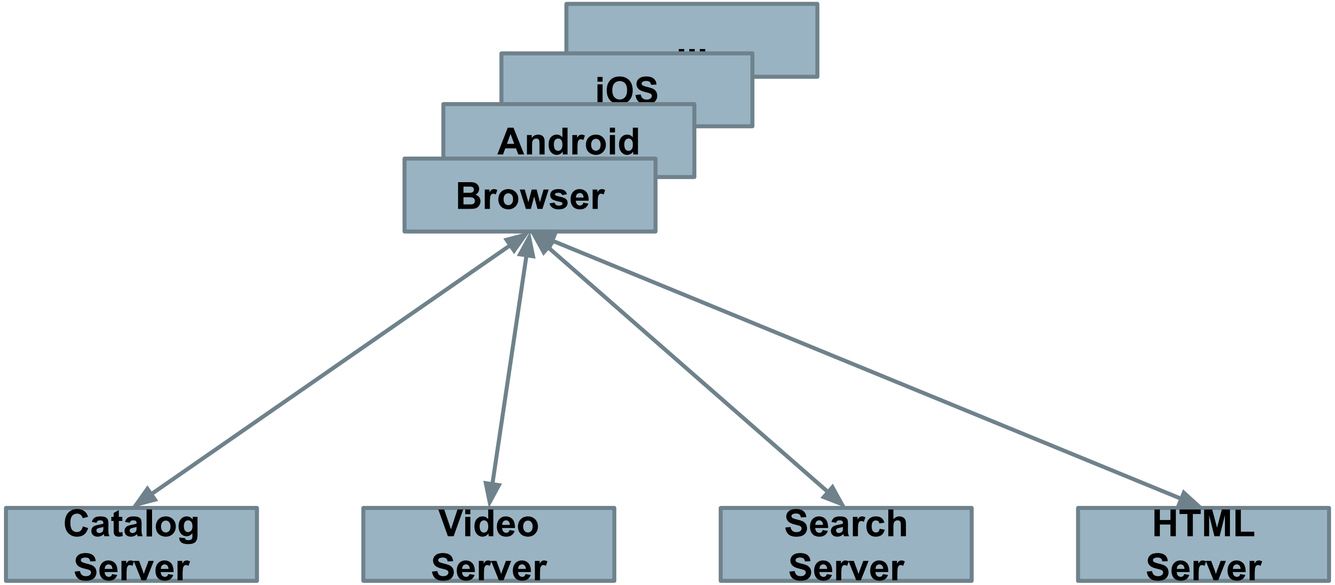
- Single point of failure.
- Components must agree on data model
  - (inevitably a compromise).
- Data evolution difficult.
- Communication may be inefficient.

Client-Server Model

Functionality organized into distributed services:

- Servers that offer services.
  - Print server, file server, code compilation server, etc..
- Clients that call on these services.
  - Through locally-installed front-end.
- Network allows clients to access services.
  - Distributed systems connected across the internet.

Film Library Example



Client-Server Model Characteristics

Advantages

- Distributed architecture.
  - Failure in one server does not impact others.
- Effective use of networked systems and their CPUs. May allow cheaper hardware.
- Easy to add new servers or upgrade existing servers.

Disadvantages

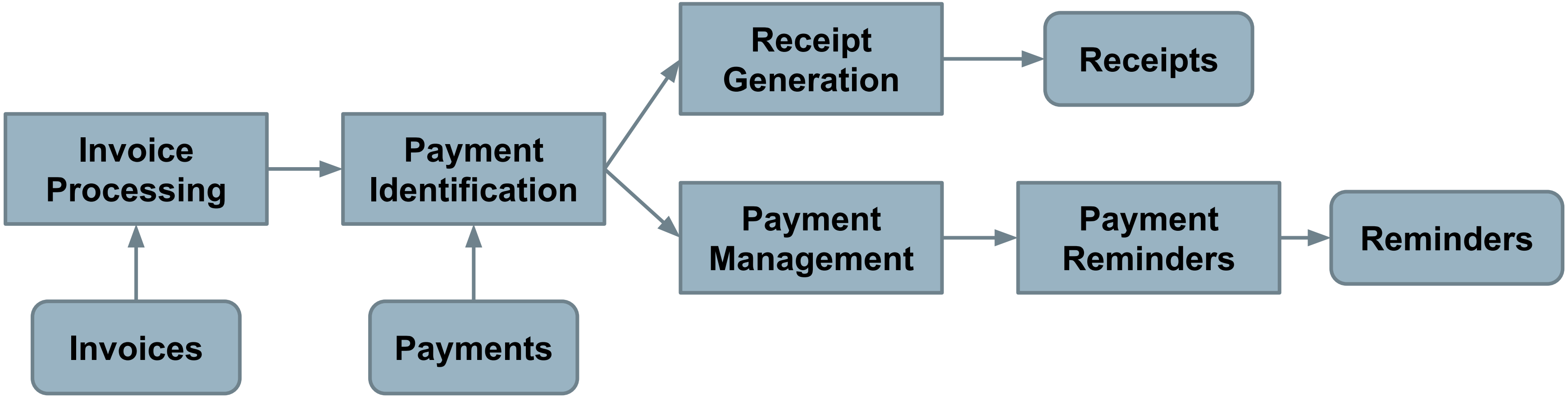
- Performance unpredictable
  - (depends on system/network)
- Each service is a point of failure.
- Data exchange may be inefficient
  - (server -> client -> server)
- Management problems if servers owned by others.

Pipe and Filter Model

Input is taken in by one component, processed, and the output serves as input to the next component.

- Each processing step transforms data.
- Transformations execute sequentially or in parallel.
- Data processed as items or batches.
- Similar to Unix command line:
  - `cat file.txt | cut -d, -f 2 | sort -n | uniq -c`

Customer Invoicing Example



Pipe and Filter Characteristics

Advantages

- Easy to understand communication.
- Supports reuse.
- Add features by adding new components to sequence.

Disadvantages

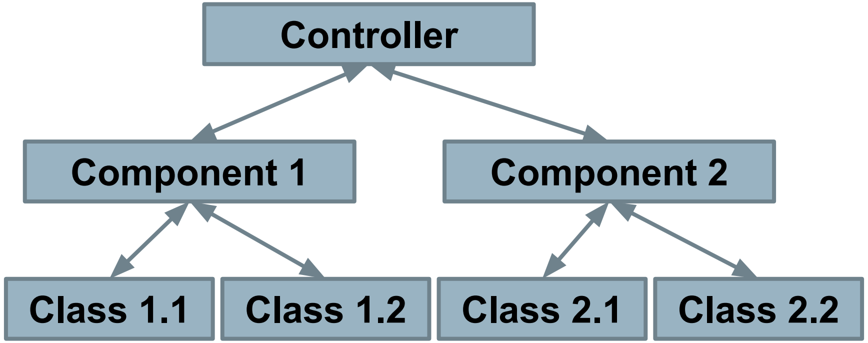
- Communication format must be agreed on.
    - Each transformation needs to accept and output right format.
- Increases overhead.
- Can hurt reuse if code doesn’t accept structure.

Dynamic Structuring

- Model control relationships between components.
- During execution, how do components work together to respond to requests?
  - **Centralized Control:**
    - One component has overall responsibility and stops/starts others.
  - **Event-Based Control:**
    - Each component can respond to events generated by others or the environment.

Centralized Control: Call-Return

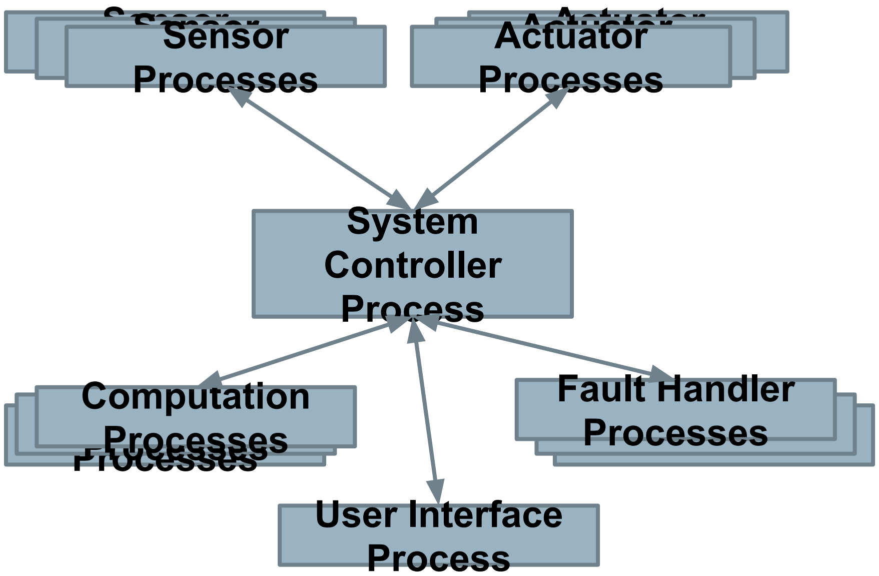
Central controller takes responsibility for managing the execution of other subsystems.



Call-Return Model

- Applicable to sequential systems.
- Top-down model: control starts at the top of hierarchy and moves downwards.

Centralized Control: Manager Model



- Applicable to concurrent systems.
- One process controls stopping, starting, and coordination of other processes.

Decentralized Control: Event-Driven

Control driven by external events where timing is out of control of components that process the event.

- **Broadcast Model**
  - An event is broadcast to all components.
  - Any that needs to respond to the event does so.
- **Interrupt-Driven Model**
  - Events processed by interrupt handler and passed to proper component for processing.

Broadcast Model

Event broadcast to all components, any that can handle it respond.

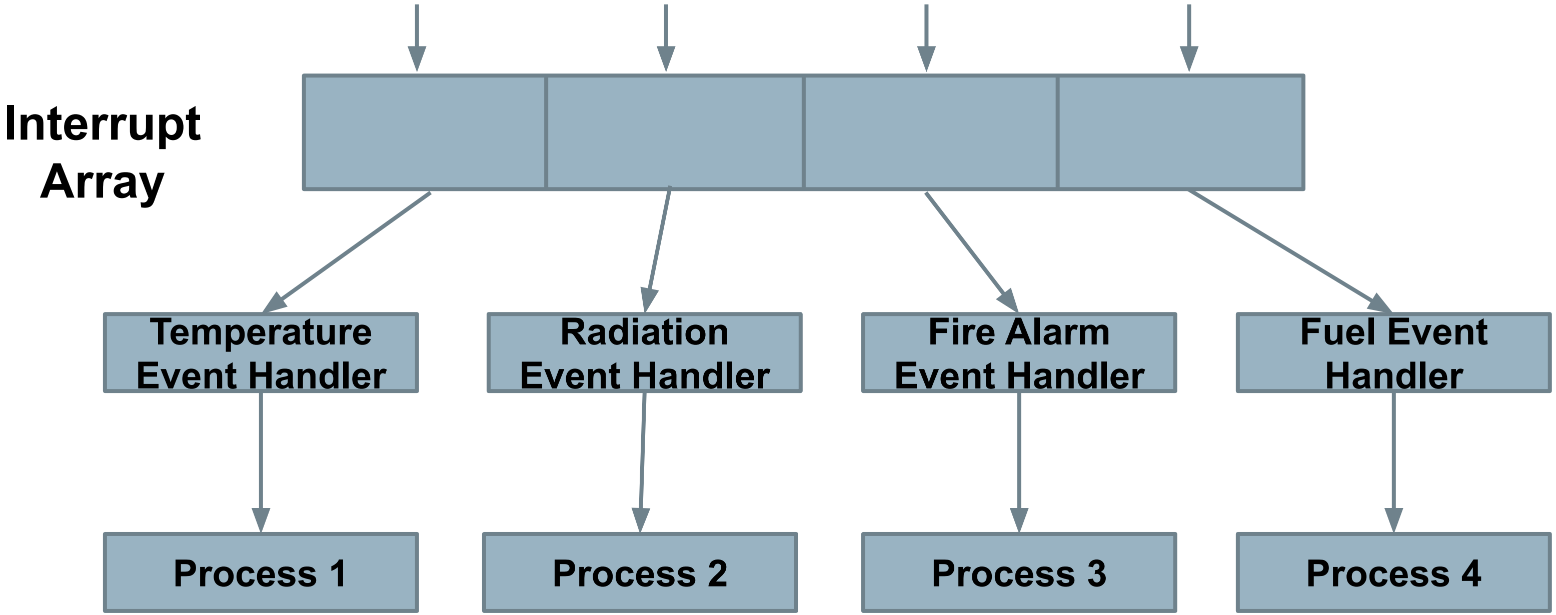
- Components register interest in specific events.
  - When these occur, control is transferred to registered components.
- Effective for distributed systems.
  - When component fails, others can potentially respond.
  - Components don’t know when or if event will be handled.

Interrupt-Driven Model

Events processed by interrupt handler and passed to components for processing.

- For each type of interrupt, handler listens for the event and coordinates response.
- Each interrupt type associated with a memory location. Handlers watch that address.
- Ensures fast response to event.
    - Complex to program, hard to validate.

Nuclear Plant Interrupt Example



We Have Learned

- Frameworks
  - Composition-based, load-time.
  - White Box: Subclass an abstract parent, override template methods with specific functionality.
  - Black Box: Register plug-in objects that provide specific functionality.
  - Provides clear modularity, but requires extensive up-front design effort.

We Have Learned

- Components
  - Standalone functionality with explicit interface and dependencies.
  - Interfaces often standardized (REST).
  - Can be reused in many projects.
  - Integrated as part of a broader architectural design.

We Have Learned

- The architecture must consider static structure, dynamic structure, externally-visible behaviors, and performance.
- Architectural models help organize a system.
  - Layered, repository, client-server, and pipe and filter models - also many others.
- Control models include centralized control and event-driven models.

Next Time

- API Design
  - REST APIs
  - API design principles
  - Designing reusable APIs

- Assignment 3 - Preprocessors - Due Sunday
- Assignment 4 - Design Patterns/Modularity
  - Due December 12

UNIVERSITY OF GOTHENBURG

CHALMERS

UNIVERSITY OF TECHNOLOGY