6. Modular Features – Handout

universität ulm
UNIVERSITÄT BERN
OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

6. Modular Features

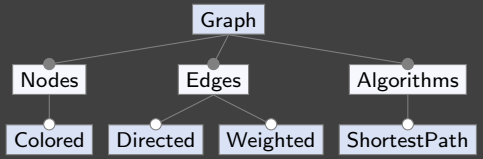
6a. Components

6b. Services and Microservices

6c. Frameworks with Plug-Ins

Recap: How to Implement Features?

Given a feature model for graphs . . .

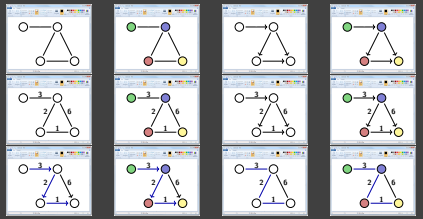


$ShortestPath \rightarrow Weighted$

. . . we can derive a valid configuration

$\{G\}$	$\{G, W\}$	$\{G, W, S\}$
$\{G, C\}$	$\{G, C, W\}$	$\{G, C, W, S\}$
$\{G, D\}$	$\{G, D, W\}$	$\{G, D, W, S\}$
$\{G, C, D\}$	$\{G, C, D, W\}$	$\{G, C, D, W, S\}$

How to Generate Products Automatically?

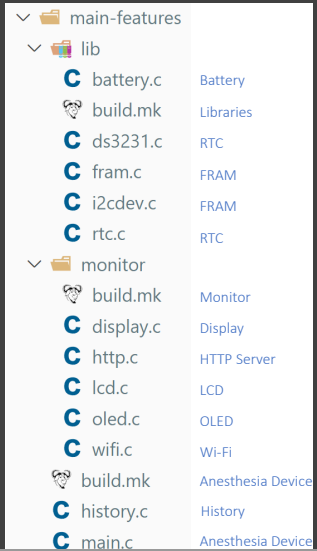


Goals

- descriptive specification of a product (i.e., a configuration, a selection of features)
- automated generation of a product with compile-time variability

Focus of Lecture 5 – Lecture 7

Recap: Features with Build Systems



Conditional Compilation with Build Systems

- exploit the expressiveness of a build system's configuration language
- include and exclude individual files or entire directories based on feature selection

Major Challenges [Lecture 5]

- build scripts may become complex, there is no limit to what can be done (e.g., you can run arbitrary shell commands on files)
  $\Rightarrow$ **hard to understand and analyze**
- no simple inclusion and exclusion of individual lines or chunks of code
  $\Rightarrow$ high-level use **only**!

Recap: Features with Preprocessors

```
class Edge {
  Node first, second;
//#ifdef Weighted
  int weight;
//#endif
  Edge(Node first, Node second) {...}
  boolean equals(Edge e) {
    return (first == e.first
      && second == e.first
//#ifndef Directed
      || first == e.second
      && second == e.first
      ) && weight == e.weight;
//#endif
  }
  void testEquality() {
    Node a = new Node();
    Node b = new Node();
    Edge e = new Edge(a, b);
    Assert.assertTrue(e.equals(e));
} }
```

Conditional Compilation with Preprocessors

- use conditional compilation facilities provided by preprocessors
- annotate and potentially remove code fragments based on feature selection

Major Challenges

[Lecture 5]

- **scattering** and **tangling**
  $\Rightarrow$ separation of concerns?
- mixes multiple languages in the same development artifact
- may **obfuscate** source code and severely impact its readability
- hard to analyze and process for existing IDEs
- often used in an ad-hoc or **undisciplined** fashion
- prone to subtle syntax, type, or runtime errors which are hard to detect [Lecture 9–Lecture 11]

Modularity

Modularization

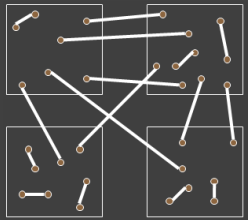
consistent application of **information hiding** and **data encapsulation** to achieve:

- strong logical connection between the inner parts of a module (high cohesion)
- precisely defined, minimal interfaces (low coupling)

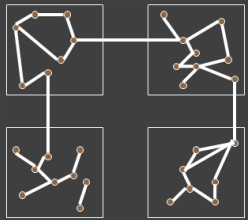
Coupling and Cohesion

- **cohesion**: measure of how well the parts of a module work together
  Rightarrow intra-module communication
- **coupling**: measure of the complexity of communication across modules
  Rightarrow inter-module communication

High Coupling, Low Cohesion



Low Coupling, High Cohesion



Why Modularity?

Traditional Reasons

- modules can be developed independently of each other (collaborative work)
- easier to maintain because changes can be made locally
- data encapsulation promotes stability and reliability
- software is easier to understand
- hiding complexity behind interfaces
- decomposition = divide and conquer

Modularization and Software Product Lines

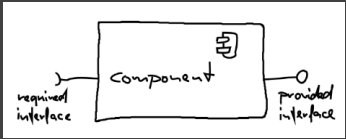
- **reuse**: parts of the software can be *reused*
- **alternatives**: modules can be *exchanged by alternative implementations*
- **variability**: modules can be *reassembled in a new context* (e.g., in other projects)

Components

Component

[Szyperski 2002]

A software component is a unit of composition with contractually specified **interfaces** and explicit **context dependencies** only. A software component can be deployed independently and is **subject to composition** by third parties.



Context/Deployment Dependencies

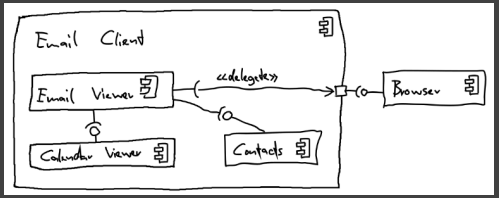
typically container or middleware (e.g., JavaEE, CORBA, OSGi, etc.)

Composition and Reuse

Components . . .

- are composed with other components to form software systems
- are supposed to be re-usable in other software systems
- may stem from third-party vendors: markets for components, make-or-buy-decisions

UML Component Diagrams



Components vs. Objects/Classes

Commonalities

Numerous similar principles:

- encapsulation and information hiding
- accessibility through public interfaces
- (de-)composition and nested objects/components
- etc.

Differences

- **objects are smaller** than components by focusing on detailed implementation problems (components aim at abstracting from implementation details)
- **object are less cohesive and stronger coupled** than components due to (deliberately) delegating lots of responsibilities to other objects whereas components aim at maximizing cohesion and minimizing coupling
- **classes are reused through inheritance and polymorphism** whereas components are reused by being integrated into a component architecture

Example: A Color Component [Apel et al. 2013]

A Reusable Component in Java

- assume that storing and printing colors is non-trivial (e.g., in our graph library)
- implement color management as a reusable component, using Java's visibility mechanism to enforce encapsulation

```
package components.color;

// public API
public class ColorComponent {
  public Color createColor(int r, int g, int b) { /* ... */ }
  public void printColor(Color color) { /* ... */ }
  public void mapColor(Object o, Color c) { /* ... */ }
  public Color getColor(Object o) { /* ... */ }

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

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

Example: A Color Component

```
public class Graph {
  private List<Node> nodes = new ArrayList<Node>();
  public Node add() {
    Node n = null;
    if (Config.COLORED) {
      Color c = ColorComponent.getInstance().createColor(0, 0, 0);
      n = new Node(nodes.size(), c);
    } else {
      n = new Node(nodes.size());
    }
    nodes.add(n);
    return n;
  }
  // ...
}
```

```
public class Node {
  private int id;
  private ColorComponent colorComp =
    ColorComponent.getInstance();
  public Node(int id) { this.id = id; }
  public Node(int id, Color c) {
    this(id);
    colorComp.mapColor(this, c);
  }
  public void print() {
    if (Config.COLORED) {
      Color c = colorComp.getColor(this);
      colorComp.printColor(c);
    }
    System.out.print(id);
  }
}
```

- we can **reuse** the color component when implementing the color feature for the graph library and also for other applications
- however: we need to write custom code to connect our implementation with the component
  $\Rightarrow$ **glue code**

Component-Based Product Lines

General Idea

- every feature is implemented by a dedicated component
- feature selection determines which components shall be integrated to form an application

Vision

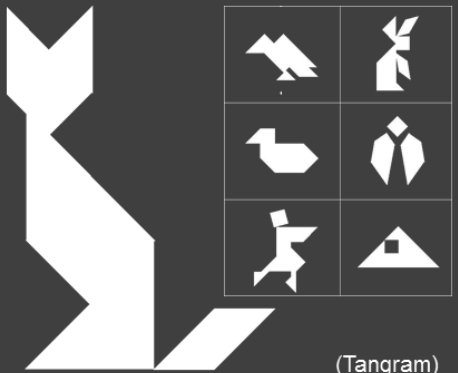
Reality

Glue Code and Customization

- developers must connect components through glue code
  exception: components are only exchanged against alternative components with identical interface
- components may contain run-time variability
  e.g., color manager in our example may be parameterized by color model RGB or CMYK

The Library Scaling Problem

Decomposition and Reuse



What is the optimal size of a component?

- large components (vertical scaling): typically not widely reusable
- small components (horizontal scaling): limited payoff for component integrators

Practical Compromise

- mostly vertically-scaled components within a few important, narrow domains (e.g., user interface construction systems)
- in other words: there is no general market for arbitrary components, but a few specialized market segments

Discussion of Components

Advantages

- supports **compile-time variability**
- modular implementation with reduced scattering and tangling (compared to runtime variability and preprocessors)

Challenges

- requires **glue code** for every product: clone-and-own for glue code? glue code with runtime variability?
- no automated generation based on feature selection
- requires preplanning to identify good level of granularity (cf. library scaling problem)
- no support for fine-granular variability $\Rightarrow$ often combined with runtime variability within components

Components – Summary

Lessons Learned

- Modularity = information hiding and data encapsulation
- Components foster a modular software architecture and design
- Reuse within and beyond product lines
- No automated product derivation, glue code is necessary

Further Reading

- Szyperski 2002
- Apel et al. 2013, Chapter 4.4

Practice

- Why is feature modeling relevant for component-based product lines?
- How can product-line engineering help to find the right trade-offs regarding the library scaling problem?

6. Modular Features

6a. Components

6b. Services and Microservices

- (Micro-)Services
- Services vs. Components
- Microservice Architectures
- Implementation of Product Lines
- Service Composition
- Summary

6c. Frameworks with Plug-Ins

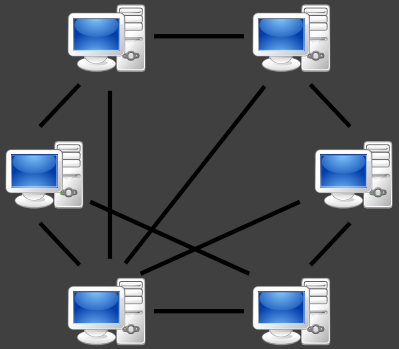
(Micro-)Services

(Micro-)Service

"[A (micro-)service is a module] implemented and operated as a small yet independent system, offering access to its internal logic and data through a well-defined network interface." Jamshidi et al., 2018

(Micro-)Service Architecture

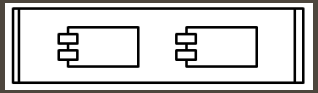
"A [micro]service is a cohesive, independent process interacting via messages. A microservice architecture [service-oriented architecture] is a distributed application where all its modules are microservices." Dragoni et al., 2017



Services vs. Components

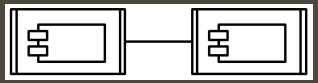
Component

Intra-process communication (i.e., method calls)



Service

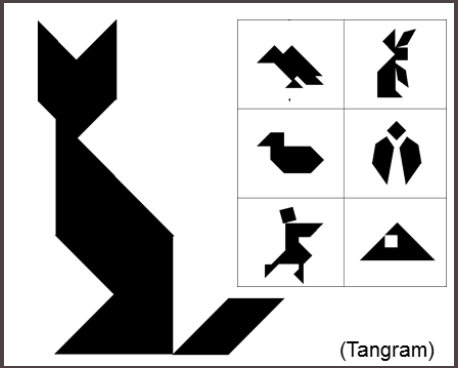
Inter-process communication (e.g., REST API)



As a consequence, each (micro-)service can be implemented using different technology stacks, whereas components are bound to the same technology (given by container or middleware).

Microservice Architecture: Motivation

Recap: The Library Scaling Problem



How small is a Microservice?

- Components are very unspecific of how to deal with the general requirement "not too small but not too big".
- On the contrary, there is a clear philosophy behind microservice architectures, largely driven by organizational constraints wrt. agile teams and continuous delivery.

"If the codebase is too big to be **managed by a small team**, looking to break it down is very sensible. [...] The smaller the service, the more you **maximize the benefits and downsides** of microservice architecture." Sam Newman, 2015

Microservice Architecture: Philosophy and Principles

Conway's Law

"Organizations which design systems [...] are constrained to produce designs which are copies of the communication structures of these organizations." Melvin Edward Conway, 1968

Single Responsibility Principle

"Gather together the things that change for the same reasons. Separate those things that change for different reasons." Robert C. Martin, 2014

"You build it, you run it." Amazon CTO Werner Vogel, 2006

Consequences

- Microservices are supposed to be split along business capabilities (e.g., purchase, sale, ...) instead of technical concerns (e.g., UI, persistence, ...)
- Each microservice is built (full stack) *and* operated by a small agile team that takes over full responsibility (cf. DevOps)

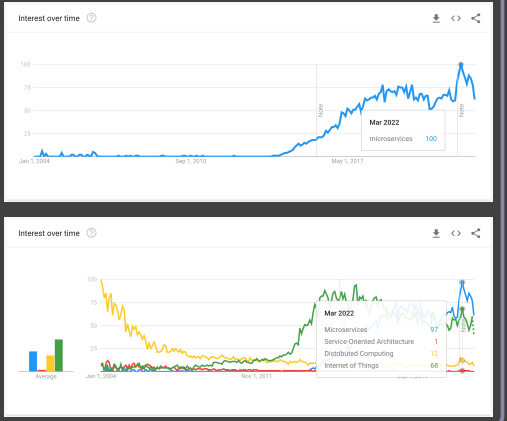
"[A microservice] could be re-written in two weeks." Jon Eaves, 2014

"Every team should be small enough that it can be fed with two pizzas." Jeff Bezos, 2018

Traditional Promises of Microservices

- **Scalability**: Microservices are small enough to be developed by a small, agile team.
- **Continuous integration/deployment**: Microservices can be deployed independently of each other.
- **Heterogeneity**: Each microservice can be implemented using its own technology stack.
- **Fault tolerance**: The crash of a single microservice should not lead to a crash of the entire system.
- **Efficiency**: Configuration of execution environment can be optimized per microservice.
- **Modernization**: A microservice can be easily replaced by an alternative one (even re-implemented from scratch).

The Microservice "Hype"



Service-Oriented Implementation of Software Product Lines

Recap: Component-Based Implementation

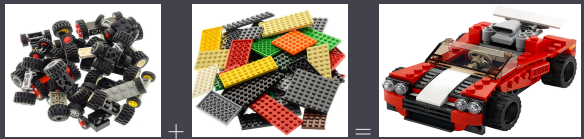
Plenty of Glue Code

Same Idea

- Features are implemented as services.
- Feature selection determines the services to be composed.

However

"Standardized" service composition instead of highly individual glue code.

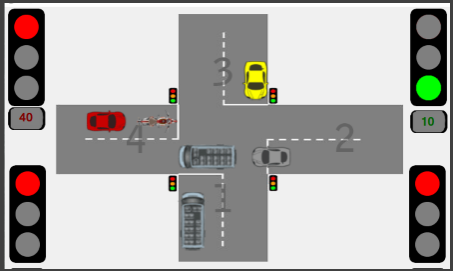


Service Composition

Orchestration

Description of an executable (business-)process as a combination of services (centralized perspective).

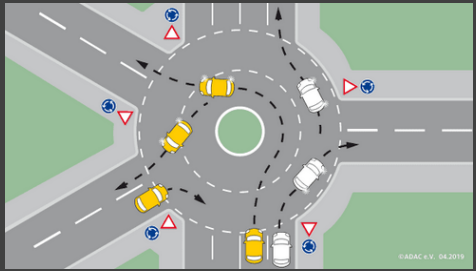
Web Services Business Process Execution Language (WS-BPEL)



Choreography

Each service describes its own task within a service composition (decentralized perspective).

Web Services Choreography Description Language (WS-CDL)



Services and Microservices – Summary

Lessons Learned

- Services are another kind of module implemented and operated independently of each other
- Microservices have a clear philosophy regarding their size, driven by organizational constraints
- Reuse within and beyond product lines
- No automated product derivation, orchestration or choreography is necessary

Further Reading

- Apel et al. 2013, Chapter 4.4

Practice

- We have talked a lot about promises of microservices. Do you also see any drawbacks?
- In a component-based product-line implementation, practitioners often rely on clone-and-own for glue code. How could we handle variability in service orchestrations?

6. Modular Features

6a. Components

6b. Services and Microservices

6c. Frameworks with Plug-Ins

- Hot Spots and Plug-Ins
- Preplanned Extensions
- Basic Design Principles
- Plug-In Loading and Management
- Frameworks in the Wild
- Implementation of Product Lines
- Discussion
- Summary
- FAQ

Framework with Plug-Ins

Framework and Hot Spot [Apel et al. 2013, pp. 80–81]

A **framework** is a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuse at a larger granularity than classes. A framework is open for **extension** at explicit **hot spots** (aka. extention point).

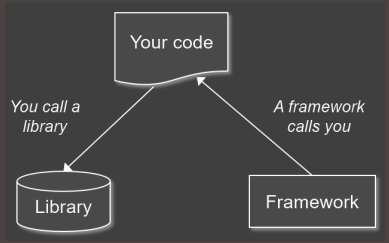
Plug-In [Apel et al. 2013, pp. 80–81]

A **plug-in** extends hot spots of a [...] framework with custom behavior. A plug-in can be separately compiled and deployed.

Frameworks with plug-ins are also called **black-box frameworks**: Developers need to understand interfaces, but not the internal framework implementation.

Inversion of Control

Hollywood principle: "Don't call us, we call you"



- Can be understood in terms of the observer and/or strategy pattern: The framework exposes explicit hot spots, at which plug-ins can register observers and strategies.
- Requires **preplanning** for possible future extensions

Real-World Example: Preplanned Bike Extensions

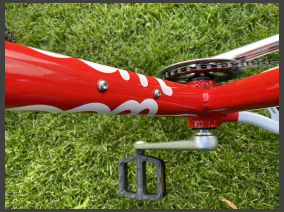
bike lock

front wheel brake

rear wheel brake

kickstand

Real-World Example: Preplanned Bike Extensions



framework with
extension points

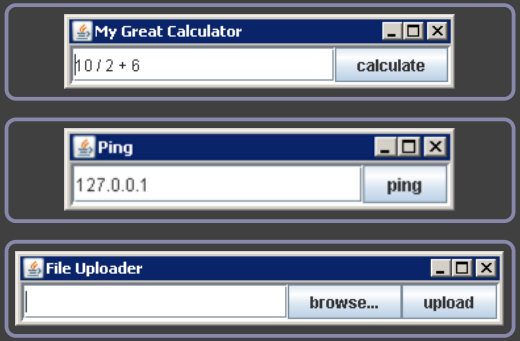
+

plug-ins

=

framework with plug-ins

Simple Java Example: A Family of Dialogs [Apel et al. 2013]



- All dialogs have a similar structure (basically textfield + button)
- Large parts of the source code are identical:
  - Main method,
  - Initialization of windows, textfield and button
  - layouting,
  - window closing and disposal,
  - etc.

Simple Java Example: A Family of Dialogs

```
public class Calc extends JFrame {
  private JTextField textfield;
  public static void main(String[] args) {
    new Calc().setVisible(true);
  }
  public Calc() { init(); }
  protected void init() {
    JPanel p = new JPanel(new BorderLayout());
    p.setBorder(new BevelBorder(/* ... */));
    JButton button = new JButton();
    button.setText("calculate");
    p.add(button, BorderLayout.EAST);
    textfield = new JTextField("");
    textfield.setText("10 / 2 + 6");
    textfield.setPreferredSize(new Dimension(350, 40));
    p.add(textfield, BorderLayout.WEST);
    button.addActionListener(new ActionListener()
      { /* calculate */ });
    this.setContentPane(p);
    this.setTitle("My Great Calculator");
    this.pack();
    // ...
  }
}
```

```
public class Ping extends JFrame {
  private JTextField textfield;
  public static void main(String[] args) {
    new Calc().setVisible(true);
  }
  public Calc() { init(); }
  protected void init() {
    JPanel p = new JPanel(new BorderLayout());
    p.setBorder(new BevelBorder(/* ... */));
    JButton button = new JButton();
    button.setText("ping");
    p.add(button, BorderLayout.EAST);
    textfield = new JTextField("");
    textfield.setText("127.0.0.1");
    textfield.setPreferredSize(new Dimension(350, 40));
    p.add(textfield, BorderLayout.WEST);
    button.addActionListener(new ActionListener()
      { /* calculate */ });
    this.setContentPane(p);
    this.setTitle("Ping");
    this.pack();
    // ...
  }
}
```

Simple Java Example: A Family of Dialogs

plug-in implementation hidden from application

```
public class Application extends JFrame {
  private Plugin plugin;
  // ...
  public Application(Plugin plugin) {
    this.plugin = plugin;
    plugin.setApplication(this);
    init();
  }
  protected void init() {
    JPanel p = new JPanel(new BorderLayout());
    p.setBorder(new BevelBorder(/*...*/));
    JButton button = new JButton();
    button.setText(plugin.getButtonText());
    p.add(button, BorderLayout.EAST);
    textfield = new JTextField("");
    textfield.setText(plugin.getInitialText());
    textfield.setPreferredSize(new Dimension(200, 20));
    p.add(textfield, BorderLayout.WEST);
    button.addActionListener(/*... @plugin.buttonClicked();@ ...*/);
    this.setContentPane(p);
    this.setTitle(plugin.getApplicationTitle());
    // ...
  }
  public String getInput() {
    return textfield.getText();
  }
}
```

```
public interface Plugin {
  String getApplicationTitle();
  String getButtonText();
  String getInitialText();
  void buttonClicked();
  void setApplication(Application app);
}
```

```
public class CalcPlugin implements Plugin {
  private Application application;

  public String getApplicationTitle() {
    return "My Great Calculator";
  }
  public String getButtonText() {
    return "calculate";
  }
  public String getInitialText() {
    return "10 / 2 + 6";
  }
  public void buttonClicked() {
    calculate(application.getInput());
  }
  public void setApplication(Application app) {
    application = app;
  }
  private void calculate(String expression) {
    /* calculate */
  }
}
```

Simple Example: A Family of Dialogs

application implementation hidden from plug-in

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

```
public interface InputProvider {
  public String getInput();
}
```

```
public interface Plugin {
  String getApplicationTitle();
  String getButtonText();
  String getInitialText();
  void buttonClicked();
  void setInputProvider(InputProvider p);
}
```

```
public class CalcPlugin implements Plugin {
  private InputProvider inputProvider;

  public String getApplicationTitle() { /*...*/ }
  public String getButtonText() { /*...*/ }
  public String getInitialText() { /*...*/ }
  public void buttonClicked() {
    calculate(inputProvider.getInput());
  }
  public void setInputProvider(InputProvider p) {
    inputProvider = p;
  }
  private void calculate(String expression) {
    /* calculate */
  }
}
```

Plug-In Loading and Management

Simple Example vs. Reality

simplification in our previous example:

- a single extension point supporting the registration of a single plug-in
- plug-in implementation known at compile-time

typical requirements in practice:

- an extension point typically supports the registration of arbitrarily many plug-ins
- a single plug-in may extend several extension points
- a plug-in may add new extension points to the framework (framework of frameworks)
- plug-in implementation provided by third parties

Plug-In Loader

- Searches in a dedicated directory for DLL/JAR/XML files
- Tests whether file implements a plug-in
- Checks dependencies
- Initializes plug-ins

Plug-In Manager

GUI and/or console interface for plug-in administration and configuration

Example: Plug-In Loading and Management

Plug-In Loader using Java Reflection

```
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);
        new Application((Plugin)
          pluginClass.newInstance()).setVisible(true);
      } catch (Exception e) {
        System.out.println(" Cannot load plugin " +
          pluginName + ", reason: " + e);
      }
    }
  }
}
```

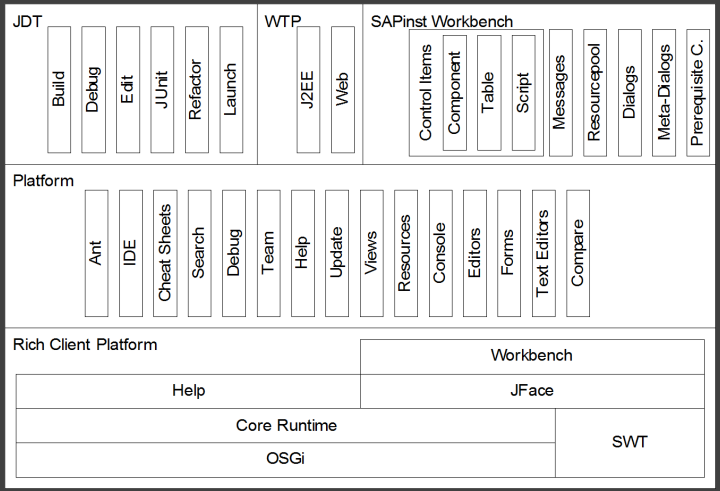
Handling multiple Plug-Ins

```
public class Application {
  private List<Plugin> plugins;

  public Application(List<Plugin> plugins) {
    this.plugins = plugins;
    for (Plugin plugin : plugins) {
      plugin.init();
    }
  }

  public Message processMsg(Message msg) {
    for (Plugin plugin : plugins) {
      msg = plugin.process(msg);
      // ...
    }
    return msg;
  }
}
```

Frameworks in the Wild: Eclipse



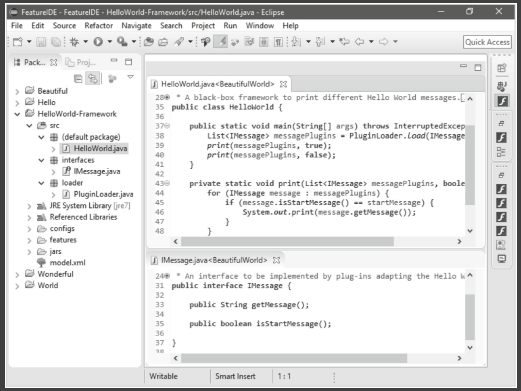
Versatile IDE

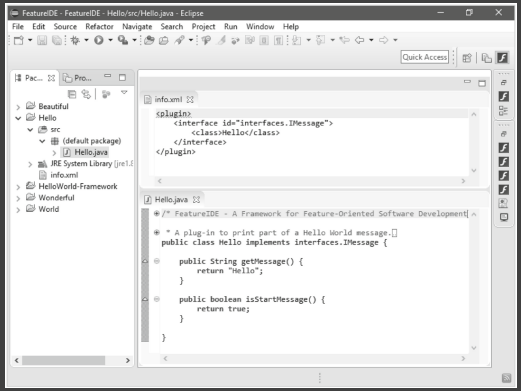
- Lots of common functionality required by any IDE (e.g., editors, incremental project build, etc.)
- Only language-specific extensions need to be registered (e.g., syntax highlighting, compiler, etc.)

Specifically in Eclipse

- Actually a set of (recursively nested) frameworks
- Largely declarative description of extension points

Frameworks in the Wild: Eclipse [Meinicke et al. 2017]



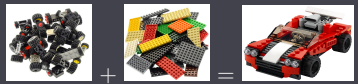


Frameworks in the Wild: Further Examples

- Other IDEs (e.g., IntelliJ, ...)
- Unit test frameworks (e.g., JUnit, ...)
- Frontend frameworks (e.g., React, Angular ...)
- Backend frameworks (e.g., Spring Boot, Ruby on Rails, Django, ...)
- Multimedia frameworks (e.g., DirectX, ...)
- Raster/vector graphics editors (e.g., Adobe Photoshop, MS Visio, ...)
- Instant messenger frameworks (e.g., Miranda, Trillian)
- Compiler frameworks (e.g., LLVM, Polyglot, abc, JustAddJ)
- Web browsers (e.g., Firefox, ...)
- E-Mail clients (e.g., Thunderbird, ...)
- etc.

Framework-Based Implementation of Software Product Lines

Recap: Service-Based Implementation



Still needs some specification of "composition" (cf. orchestration vs. choreography)

Same Idea

- Features are implemented by different plug-ins
- Feature selection determines the plug-ins to be loaded and registered

neither glue code nor explicit service composition required

full automation comes at a price (cf. preplanning problem)

Example: Extending Basic Graphs by Plug-Ins?

```
public class Graph {
  private List<GraphPlugin> plugins = new ArrayList<GraphPlugin>();
  // ...
  public void registerPlugin(GraphPlugin p){
    plugins.add(p);
  }
  public void addNode(int id, Color c){
    Node n = new Node(id);
    notifyAdd(n, c);
    nodes.add(n);
  }
  public void print() {
    for (Node n : nodes) {
      notifyPrint(n);
      // ...
    }
    // ...
  }
  private void notifyAdd(Node n, Color c) {
    for (GraphPlugin p : plugins) {
      p.aboutToAdd(n, c);
    }
  }
  private void notifyPrint(Node n) {
    for (GraphPlugin p : plugins) {
      p.aboutToPrint(n);
    }
  }
  // ...
}
```

```
public interface GraphPlugin {
  public void aboutToAdd(Node n, Color c);
  public void aboutToAdd(Edge e, Weight w);
  public void aboutToPrint(Node n);
  public void aboutToPrint(Edge e);
}
```

```
public class ColorPlugin implements GraphPlugin {
  private Map<Node, Color> map = new HashMap<Node, Color>();

  public void aboutToAdd(Node n, Color c) {
    map.put(n, c);
  }

  public void aboutToAdd(Edge e, Weight w) {
    // do nothing~
  }

  public void aboutToPrint(Node n) {
    Color c = map.get(n);
    Color.setDisplayColor(c);
  }

  public void aboutToPrint(Edge e) {
    // do nothing~
  }
}
```

Challenges and Problems

In our example, we can observe that:

- There are lots of empty methods in the ColorPlugin
- The Framework consults all registered plug-ins before printing a node or edge

General Challenge: Cross-cutting Concerns

Implementing cross-cutting concerns as plug-ins

- typically leads to huge interfaces, large parts of which are irrelevant for a dedicated plug-in
- causes lots of communication overhead between plug-ins and framework

If we were not familiar with our graph library, would we anticipate that:

- Colors and weights should be part of the Plugin interface?
- Every plug-in needs to be notified that the framework is about to print a node or edge?

Generally known as Preplanning Problem

- Hard to identify and foresee the relevant hot spots and nature of extensions
- Developing a framework needs lots of expertise and excellent domain knowledge

Frameworks with Plug-Ins – Summary

Lessons Learned

- A framework is open to extension by integrating plug-ins at explicit hot spots.
- The framework takes full control over all plug-ins (load-time and run-time).
- Enables full automation but requires preplanning effort.

Further Reading

- Apel et al. 2013, Chapter 4.3

Practice

- What are possible extensions for a bike?
- How are bike extensions affected by the preplanning problem?

FAQ – 6. Modular Features

Lecture 6a

- What are problems of previous implementation techniques?
- How to implement product lines with components?
- What is modularity, cohesion, coupling, glue code? Why does it matter?
- Why is it hard to find the right size for components?
- What is the library scaling problem? How can product-line engineering help?
- How to develop new features or variants? Exemplify!
- What are (dis)advantages of components?
- When (not) to implement product lines with components?

Lecture 6b

- What is a (micro-)service, orchestration, choreography?
- What are commonalities and differences between components and services?
- How to implement product lines with (micro-)services?
- How are microservices related to Conway's law, single responsibility principle, agile development, DevOps?
- How to compose services?
- How to develop new features or variants? Exemplify!
- What are (dis)advantages of services?
- When (not) to implement product lines with services?

Lecture 6c

- What are (black-box) frameworks, plug-ins, extension points/hot spots, extensions?
- What is inversion of control? Why is it useful?
- How to hide implementation details from both, framework and plug-ins? Why is it useful?
- How to implement product lines with plug-ins?
- How to develop new features or variants? Exemplify!
- What are (dis)advantages of plug-ins?
- When (not) to implement product lines with plug-ins?