Part I: Ad-Hoc Approaches for Variability

1. Introduction
2. **Runtime Variability and Design Patterns**
3. Compile-Time Variability with Clone-and-Own

Part II: Modeling & Implementing Features

4. Feature Modeling
5. Conditional Compilation
6. Modular Features
7. Languages for Features
8. Development Process

Part III: Quality Assurance and Outlook

9. Feature Interactions
10. Product-Line Analyses
11. Product-Line Testing
12. Evolution and Maintenance

2a. Configuration of Runtime Variability

- Variability and Binding Time
- Command-Line Parameters
- Preference Dialogs
- Configuration Files
- Runtime Variability
- Configuration Options
- Valid Combinations of Options
- Summary

2b. Realization of Runtime Variability

- Recap and Motivating Example
- Global Variables
- Method Parameters
- Reconfiguration at Runtime?
- Code Scattering, Tangling, and Replication
- Summary

2c. Design Patterns for Variability

- Recap on Object-Orientation and Design Patterns
- Template Method Pattern
- Abstract Factory Pattern
- Decorator Pattern
- Trade-Offs and Limitations
- Summary
- FAQ

2. Runtime Variability and Design Patterns – Handout

universität ulm

UNIVERSITÄT BERN

OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

2. Runtime Variability and Design Patterns

2a. Configuration of Runtime Variability

Variability and Binding Time

Command-Line Parameters

Preference Dialogs

Configuration Files

Runtime Variability

Configuration Options

Valid Combinations of Options

Summary

2b. Realization of Runtime Variability

2c. Design Patterns for Variability

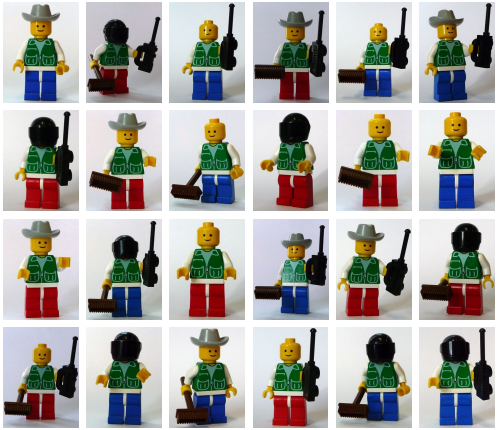
Recap: Software Product Lines

Software Product Line [Northrop et al. 2012, p. 5]

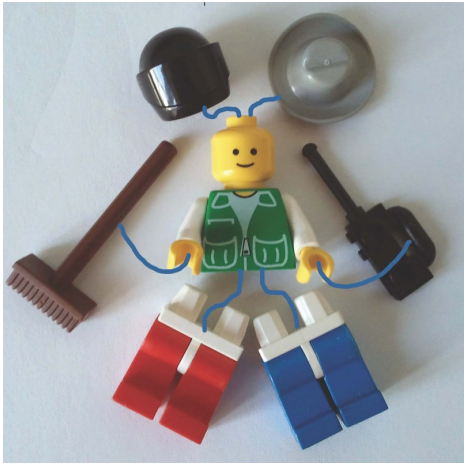
"A **software product line** is

- a set of software-intensive systems (aka. products or variants)
- that share a common, managed set of features
- satisfying the specific needs of a particular market segment or mission (aka. domain)
- and that are developed from a common set of core assets in a prescribed way."

[Software Engineering Institute, Carnegie Mellon University]



How to Implement Software Product Lines?



Key Issues

- Systematic reuse of implementation artifacts
- Explicit handling of variability

Variability [Apel et al. 2013, p. 48]

"**Variability** is the ability to derive different products from a common set of artifacts."

Variability-Intensive System

Any software product line is a variability-intensive system.

Variability and Binding Times

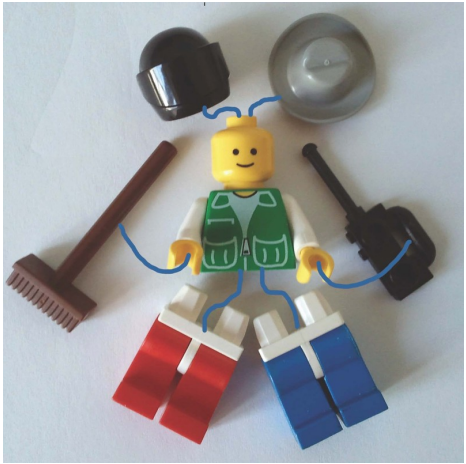
Binding Time [Apel et al. 2013, p. 48]

- Variability offers choices
- Derivation of a product requires to make decisions (aka. binding)
- Decisions may be bound at different binding times

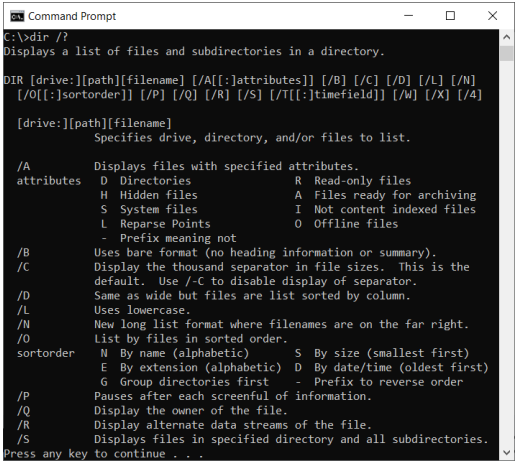
When? By whom? How?

Lecture 2a: **when** and **by whom**

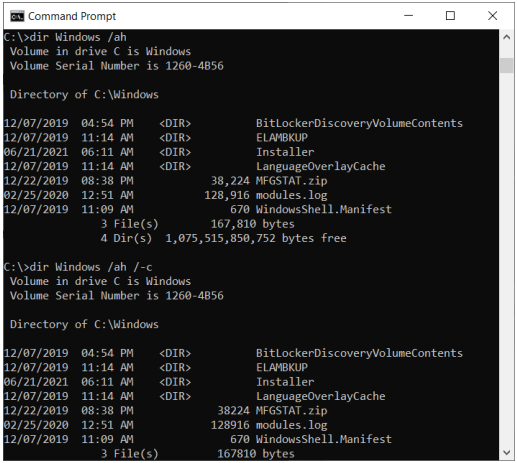
Lecture 2b: **how**



Example: Command-Line Parameters

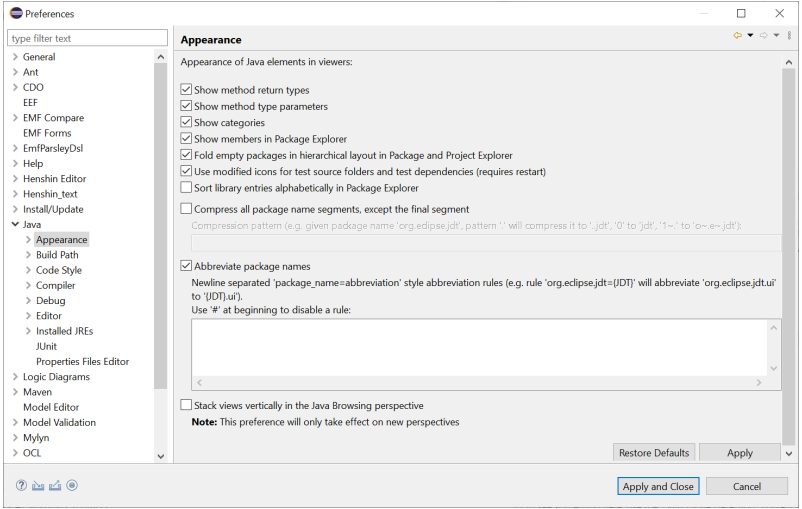


Description of configuration options

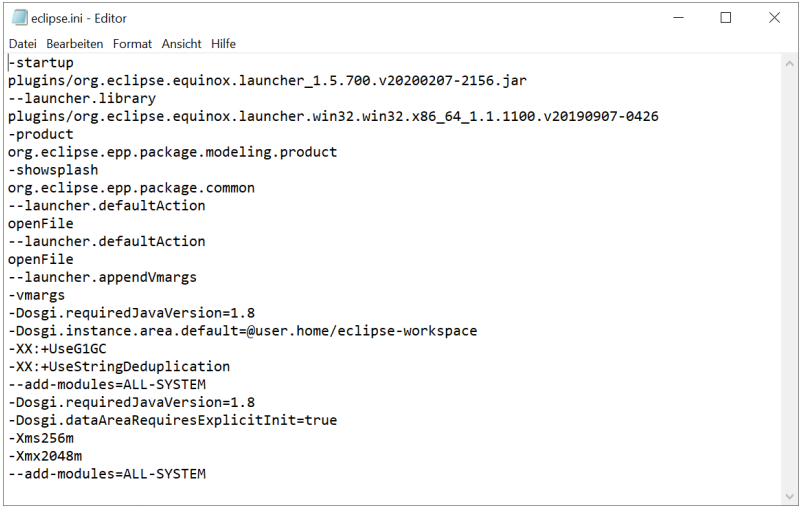


Two different instances? separator , in file sizes

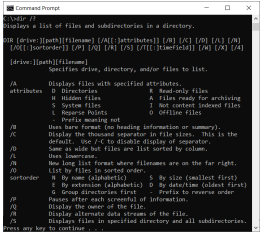
Example: Preference Dialogs

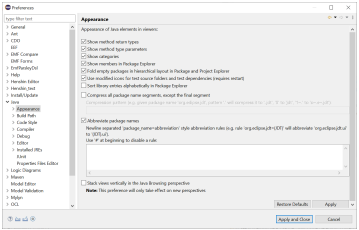


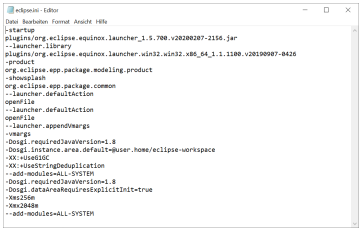
Example: Configuration Files



What do these examples have in common?







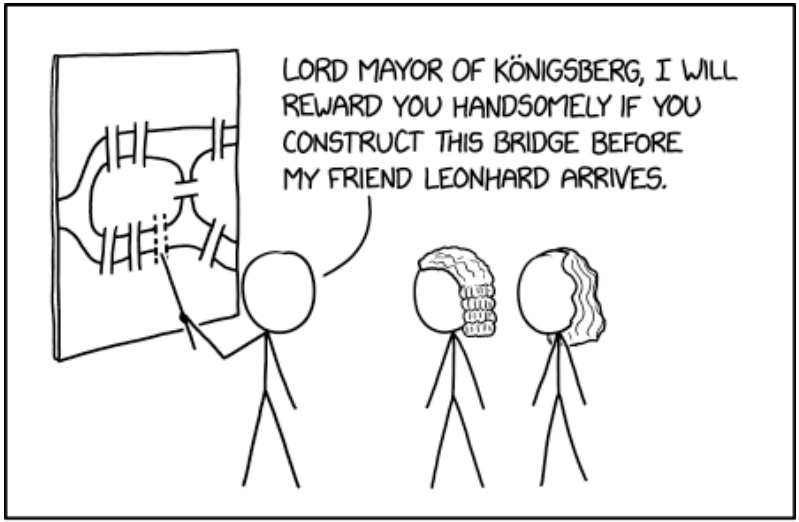
Configuration Options

- Behavior of a program is determined by configuration options being interpreted at runtime
- Choices offered by variability are decided at runtime
- Configuration may happen interactively (command-line parameters, preference dialogs) or non-interactively (configuration files)

Runtime Variability

[Apel et al. 2013, p. 49]

Runtime variability is decided after compilation when the program is started (aka. load-time variability) or during program execution.



I TRIED TO USE A TIME MACHINE TO CHEAT ON MY ALGORITHMS FINAL BY PREVENTING GRAPH THEORY FROM BEING INVENTED.

Example: Graph Library

A simple library for graphs providing ...

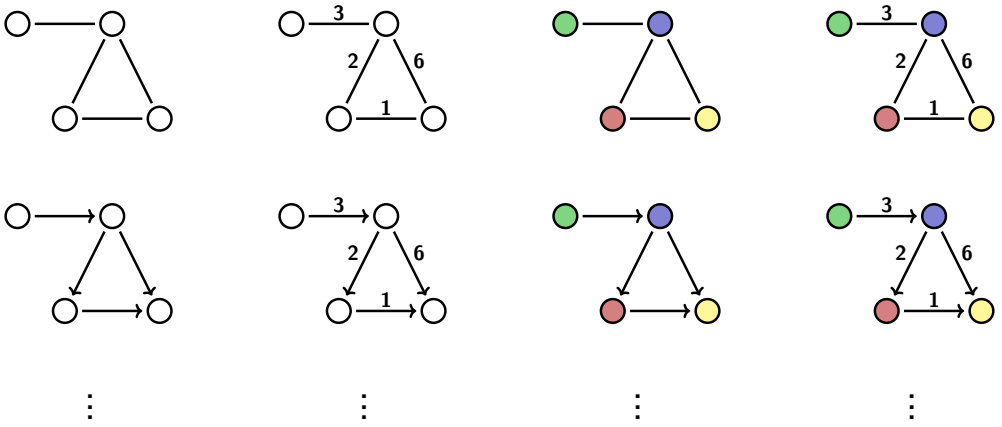
... Data Structures

- Directed/undirected edges
- Weighted/unweighted edges
- Colored/uncolored nodes
- ...

... and Algorithms

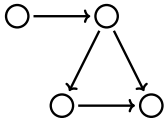
- Vertex numbering
- Vertex coloring
- Shortest path
- Minimum spanning tree
- ...

Features of a Graph

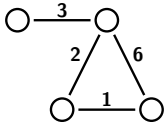


Features as Configuration Options

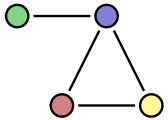
Directed



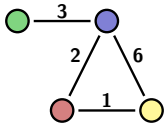
Weighted

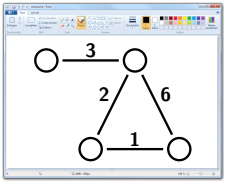


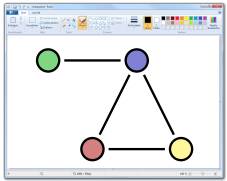
Colored



Weighted, Colored







Configuration of Graph Data Structures

- Typically, configuration options are **flags**
- Their boolean value determines which features are **activated** / **deactivated**

Valid Combinations of Options

Algorithm	Graph type	Weights	Coloring
Vertex numbering	*	*	*
Vertex coloring	undirected	*	colored
Shortest path	directed	weighted	*
Minimum spanning tree	undirected	weighted	*
...	...	...	...

Dependencies Between Features Must Be Checked

- when configuration options are loaded at startup
- whenever options are loaded/changed at runtime

Directed

Weighted

Colored

Weighted, Colored

Configuration of Runtime Variability – Summary

Lessons Learned

- External setting of configuration options through command-line parameters, preference dialogs, configuration files
- Validity of combinations may be affected by dependencies between features

Further Reading

- Apel et al. 2013, Chapter 3, pp. 47–49
  — brief introduction of binding times

Practice

- Do you know any practical examples making use of runtime variability?
- How does the configuration take place?
- Is the configuration checked for validity?

2. Runtime Variability and Design Patterns

2a. Configuration of Runtime Variability

2b. Realization of Runtime Variability

- Recap and Motivating Example
- Global Variables
- Method Parameters
- Reconfiguration at Runtime?
- Code Scattering, Tangling, and Replication
- Summary

2c. Design Patterns for Variability

Recap: Variability and Binding Times

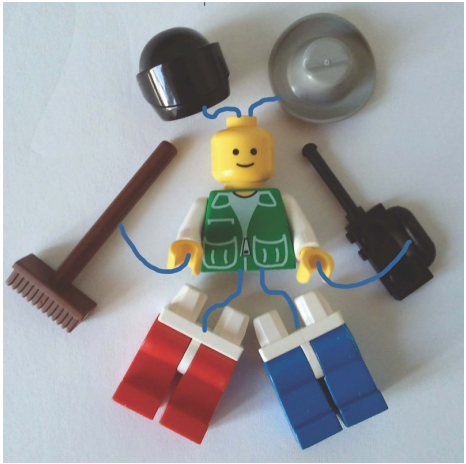
Binding Time [Apel et al. 2013, p. 48]

- Variability offers choices
- Derivation of a product requires to make decisions (aka. binding)
- Decisions may be bound at different binding times

When? By whom? How?

Lecture 2a: **when** and **by whom**

Lecture 2b: **how**



A Non-Variable Graph Implementation

```
class Graph {
  List nodes = new ArrayList();
  List edges = new ArrayList();

  Edge add(Node n, Node m) {
    Edge e = new Edge(n, m);
    e.weight = new Weight();
    nodes.add(n); nodes.add(m); edges.add(e);
    return e;
  }
  Edge add(Node n, Node m, Weight w) {
    Edge e = new Edge(n, m);
    e.weight = w;
    nodes.add(n); nodes.add(m); edges.add(e);
    return e;
  }
  void print() {
    for (int i = 0; i < edges.size(); i++) {
      ((Edge) edges.get(i)).print();
    }
  }
}
```

```
class Node {
  int id = 0;
  Color color = new Color();

  void print() {
    Color.setDisplayColor(color);
    System.out.print(id);
  }
}
```

```
class Color {
  static void setDisplayColor
    (Color c) {...}
}
```

```
class Edge {
  Node a, b;
  Weight weight;

  Edge(Node a, Node b) {
    this.a = a; this.b = b;
  }
  void print() {
    a.print(); b.print();
    weight.print();
  }
}
```

```
class Weight {
  void print() {...}
}
```

“Symbolic” Feature Traces

```
class Graph {
  List nodes = new ArrayList();
  List edges = new ArrayList();

  Edge add(Node n, Node m) {
    Edge e = new Edge(n, m);
    e.weight = new Weight();
    nodes.add(n); nodes.add(m); edges.add(e);
    return e;
  }
  Edge add(Node n, Node m, Weight w) {
    Edge e = new Edge(n, m);
    e.weight = w;
    nodes.add(n); nodes.add(m); edges.add(e);
    return e;
  }
  void print() {
    for (int i = 0; i < edges.size(); i++) {
      ((Edge) edges.get(i)).print();
    }
  }
}
```

```
class Node {
  int id = 0;
  Color color = new Color();

  void print() {
    Color.setDisplayColor(color);
    System.out.print(id);
  }
}
```

```
class Color {
  static void setDisplayColor
    (Color c) {...}
}
```

```
class Edge {
  Node a, b;
  Weight weight;

  Edge(Node a, Node b) {
    this.a = a; this.b = b;
  }
  void print() {
    a.print(); b.print();
    weight.print();
  }
}
```

```
class Weight {
  void print() {...}
}
```

Adding Variability

The Basic Idea

Conditional statements . . .

controlled by configuration options

```
class Graph {
  ...
  Edge add(Node n, Node m) {
    Edge e = new Edge(n, m);
    if (WEIGHTED) { e.weight = new Weight(); }
    nodes.add(n); nodes.add(m); edges.add(e);
    return e;
  }
  Edge add(Node n, Node m, Weight w) {
    if (!WEIGHTED) { throw new RuntimeException(); }
    Edge e = new Edge(n, m);
    e.weight = w;
    nodes.add(n); nodes.add(m); edges.add(e);
    return e;
  }
}
```

```
class Node {
  Color color;
  ...
  Node() {
    if (COLORED) { color = new Color(); }
  }
  void print() {
    if (COLORED) { Color.setDisplayColor(color); }
    System.out.print(id);
  }
}
```

```
class Edge {
  Weight weight;
  ...
  Edge(Node a, Node b) {
    this.a = a; this.b = b;
  }
  void print() {
    a.print(); b.print();
    if (WEIGHTED) { weight.print(); }
  }
}
```

Global Variables

```
public class Config {
  public static boolean COLORED = true;
  public static boolean WEIGHTED = false;
}
```

```
class Graph {
  ...
  Edge add(Node n, Node m) {
    Edge e = new Edge(n, m);
    if (Config.WEIGHTED) { e.weight = new Weight(); }
    nodes.add(n); nodes.add(m); edges.add(e);
    return e;
  }
  Edge add(Node n, Node m, Weight w) {
    if (!Config.WEIGHTED) {
      throw new RuntimeException();
    }
    Edge e = new Edge(n, m);
    e.weight = w;
    nodes.add(n); nodes.add(m); edges.add(e);
    return e;
  }
}
```

```
class Node {
  Color color;
  ...
  Node() {
    if (Config.COLORED) { color = new Color(); }
  }
  void print() {
    if (Config.COLORED) {
      Color.setDisplayColor(color);
    }
    System.out.print(id);
  }
}
```

```
class Edge {
  Weight weight;
  ...
  Edge(Node a, Node b) {
    this.a = a; this.b = b;
  }
  void print() {
    a.print(); b.print();
    if (Config.WEIGHTED) { weight.print(); }
  }
}
```

Special Case: Immutable Global Variables

```
public class Config {
  public static final boolean COLORED = true;
  public static final boolean WEIGHTED = false;
}
```

```
class Graph {
  ...
  Edge add(Node n, Node m) {
    Edge e = new Edge(n, m);
    if (Config.WEIGHTED) { e.weight = new Weight(); }
    nodes.add(n); nodes.add(m); edges.add(e);
    return e;
  }
  Edge add(Node n, Node m, Weight w) {
    if (!Config.WEIGHTED) { throw new RuntimeException(); }
    Edge e = new Edge(n, m);
    e.weight = w;
    nodes.add(n); nodes.add(m); edges.add(e);
    return e;
  }
}
```

Idea

Use static configuration when configuration parameters are known at compile time

Discussion

Advantage: Compiler optimizations may remove dead code

Disadvantage: No external configuration by the end-user

Method Parameters

Idea

- A class exposes its configuration parameters as part of its interface (i.e., method parameters)
- Parameter values are passed along with each method invocation

```
class Graph {
  boolean weighted, colored;
  ...
  Graph(boolean weighted, boolean colored) {
    this.weighted = weighted; this.colored = colored;
  }
  Edge add(Node n, Node m) {
    Edge e = new Edge(n, m, weighted);
    if (weighted) { e.weight = new Weight(); }
    nodes.add(n); nodes.add(m); edges.add(e);
    return e;
  }
}
```

```
class Edge {
  boolean weighted;
  Weight weight;
  ...
  Edge(Node a, Node b, boolean weighted) {
    this.a = a; this.b = b;
    this.weighted = weighted;
  }
  void print() {
    a.print(); b.print();
    if (weighted) { weight.print(); }
  }
}
```

Discussion

Advantage: Different instantiations (e.g., colored and uncolored graphs) within the same program

Disadvantage: May lead to methods with many parameters (code smell!)

Reconfiguration at Runtime?

```
public class Config {
  public static boolean COLORED = false;
  public static boolean WEIGHTED = false;
}
```

```
public class Node {
  Color color;
  ...
  Node(){
    if (Config.COLORED) {
      color = new Color();
    }
  }
  void print() {
    if (Config.COLORED) {
      Color.setDisplayColor(color);
    }
    System.out.print(id);
  }
}
```

Idea

Alter feature selection without stopping and restarting the program

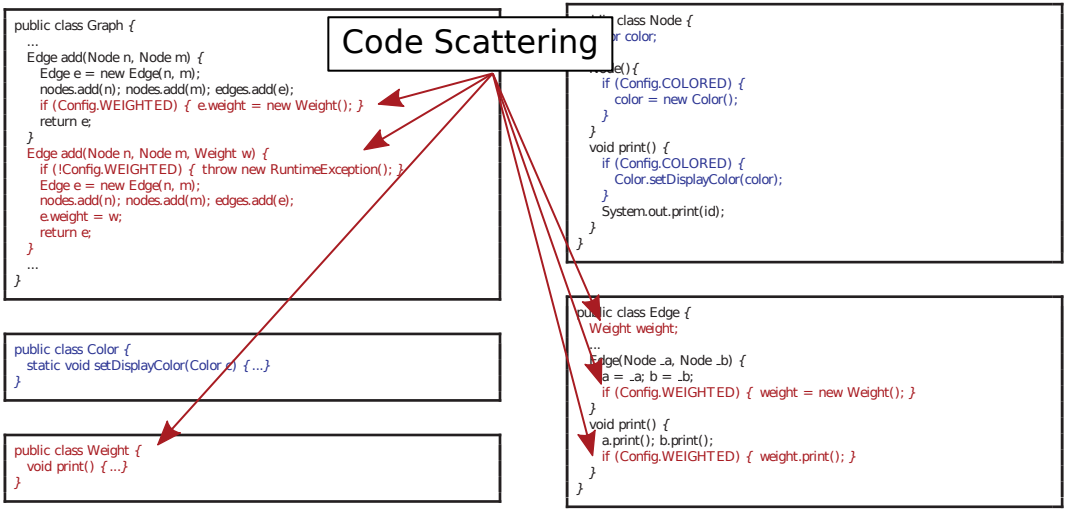
Though Experiment

What happens when we change the value of `COLORED` from `false` to `true` (at runtime)?

Discussion

- Feature-specific code may depend on certain initialization steps or assume certain invariants
- Just updating the values of configuration options does not update the current state of the program (source of bugs!)

Problem: Code Scattering



Problem: Code Tangling

```
public class Graph {
  ...
  Edge add(Node n, Node m) {
    Edge e = new Edge(n, m);
    nodes.add(n); nodes.add(m); edges.add(e);
    if (Config.WEIGHTED) { e.weight = new Weight(); }
    return e;
  }
  Edge add(Node n, Node m, Weight w) {
    if (!Config.WEIGHTED) { throw new RuntimeException(); }
    Edge e = new Edge(n, m);
    nodes.add(n); nodes.add(m); edges.add(e);
    e.weight = w;
    return e;
  }
  ...
}
```

```
public class Color {
  static void setDisplayColor(Color c) {...}
}
```

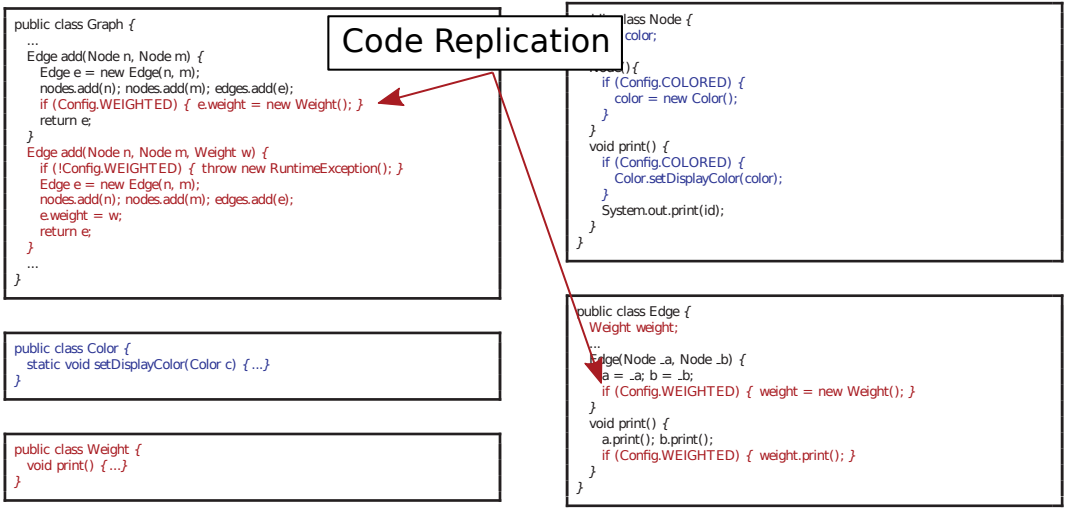
```
public class Weight {
  void print() {...}
}
```

Code Tangling

```
public class Node {
  Color color;
  ...
  Node(){
    if (Config.COLORED) {
      color = new Color();
    }
  }
  void print() {
    if (Config.COLORED) {
      Color.setDisplayColor(color);
    }
    System.out.print(id);
  }
}
```

```
public class Edge {
  Weight weight;
  ...
  Edge(Node _a, Node _b) {
    a = _a; b = _b;
    if (Config.WEIGHTED) { weight = new Weight(); }
  }
  void print() {
    a.print(); b.print();
    if (Config.WEIGHTED) { weight.print(); }
  }
}
```

Problem: Code Replication (aka. Code Clones)



Realization of Runtime Variability – Summary

Lessons Learned

- Global (immutable) variables or (lengthy) parameter lists
- Reconfiguration at runtime is possible (in principle)
- Variability is spread over the entire program
- Variable parts are always delivered

Further Reading

- Apel et al. 2013, Chapter 4
- Meinicke et al. 2017, Section 17.1

Practice

- Why are code scattering, tangling and replication problematic?
- What are the problems of variable parts being always delivered?

2. Runtime Variability and Design Patterns

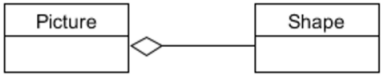
2a. Configuration of Runtime Variability

2b. Realization of Runtime Variability

2c. Design Patterns for Variability

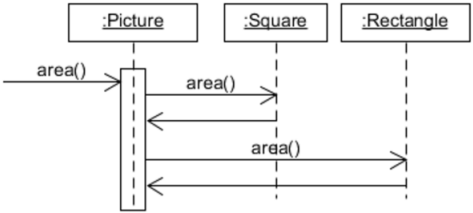
- Recap on Object-Orientation and Design Patterns
- Template Method Pattern
- Abstract Factory Pattern
- Decorator Pattern
- Trade-Offs and Limitations
- Summary
- FAQ

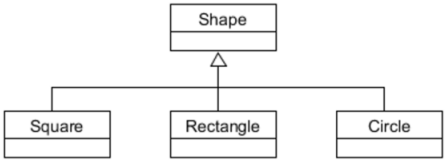
Recap: Object Orientation



Key Concepts

- **Encapsulation**:
  abstraction and information hiding
- **Composition**:
  nested objects
- **Message Passing**:
  delegating responsibility
- **Distribution of Responsibility**:
  separation of concerns
- **Inheritance**:
  conceptual hierarchy, polymorphism, reuse





Recap: Design Patterns [Gang of Four]

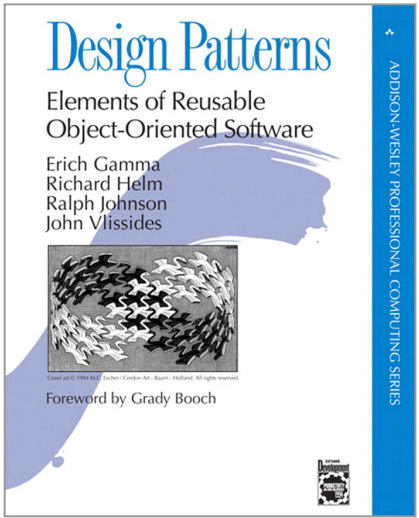
Design Patterns

- Document common solutions to concrete yet frequently occurring design problems
- Suggest a concrete implementation for a specific object-oriented programming problem

Design Patterns for Variability

Many Gang of Four (GoF) design patterns for designing software around stable abstractions and interchangeable (i.e., variable) parts, e.g.

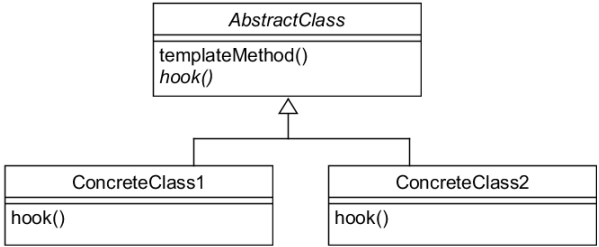
- Template Method
- Abstract Factory
- Decorator



Template Method Pattern

Template Method [Gang of Four, pp. 325–330]

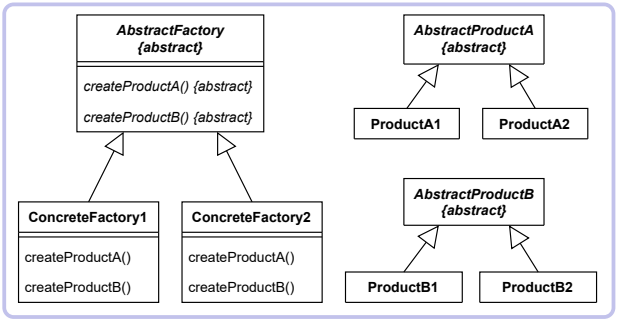
- **Intent:** "Define the overall structure of an algorithm, while allowing subclasses to refine, or redefine, certain steps."
- **Motivation:** Avoid code replication by implementing the general workflow of an algorithm once, while allowing for necessary variations.
- **Idea:** A template method defines the skeleton of an algorithm. Concrete methods override the hook methods.



Abstract Factory Pattern

Abstract Factory [Gang of Four, pp. 87–95]

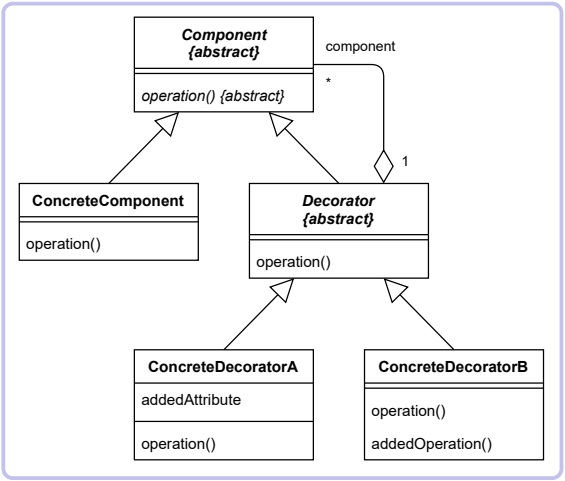
- **Intent:** "Provide an interface for creating families of related or dependent objects without specifying their concrete classes."
- **Motivation:** Avoid case distinctions when creating objects of certain kind, consistently create objects of a particular kind.
- **Idea:** Create classes for the consistent creation of objects.



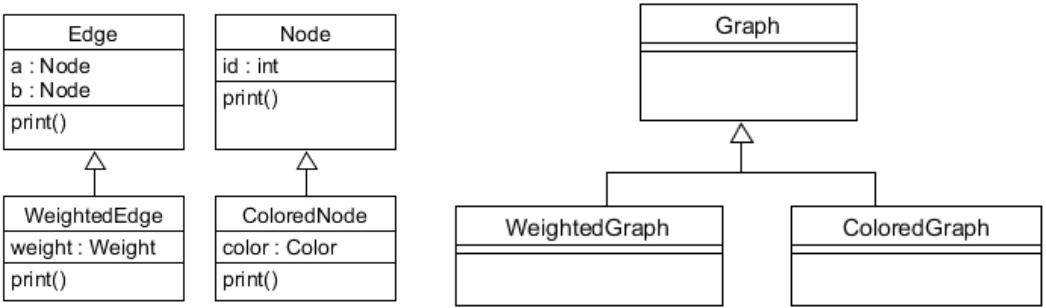
Decorator Pattern

Decorator [Gang of Four, pp. 175–184]

- **Intent:** "Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality."
- **Motivation:** Avoid explosion of static classes when combining all additional behaviors with all applicable classes.
- **Idea:** Create decorators and components with the same interface, whereas decorators forward behavior whenever feasible.



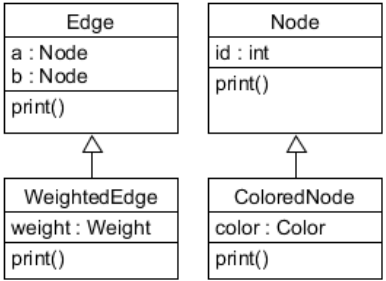
Object-Oriented Design of our Graph Library



Instantiation Through Template Method Pattern

```
class Graph {
  ...
  Edge add(Node n, Node m) {
    Edge e = createEdge();
    nv.add(n); nv.add(m); ev.add(e);
    return e;
  }
  // hook method (with default implementation)
  Edge createEdge(Node n, Node m) {
    return new Edge(n, m);
  }
}
```

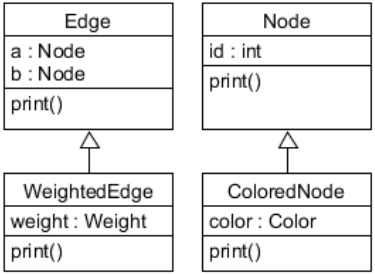
```
class WeightedGraph extends Graph {
  ...
  // override hook method
  Edge createEdge(Node n, Node m) {
    Edge e = new WeightedEdge(n, m);
    e.weight = new Weight();
    return e;
  }
}
```



Instantiation Through Abstract Factory Pattern

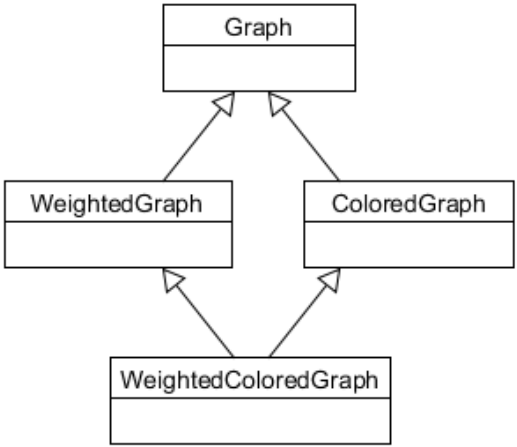
```
class Graph {
  EdgeFactory edgeFactory;
  ...
  Graph(EdgeFactory edgeFactory) {
    this.edgeFactory = edgeFactory;
  }
  Edge add(Node n, Node m) {
    Edge e = edgeFactory.createEdge(n, m);
    nodes.add(n); nodes.add(m); edges.add(e);
    return e;
  }
}
```

```
class EdgeFactory {
  Edge createEdge(Node a, Node b) {
    return new Edge(a, b);
  }
}
```




```
class WeightedEdgeFactory extends EdgeFactory {
  Edge createEdge(Node a, Node b) {
    Edge e = new WeightedEdge(n, m);
    e.weight = new Weight();
    return e;
  }
}
```

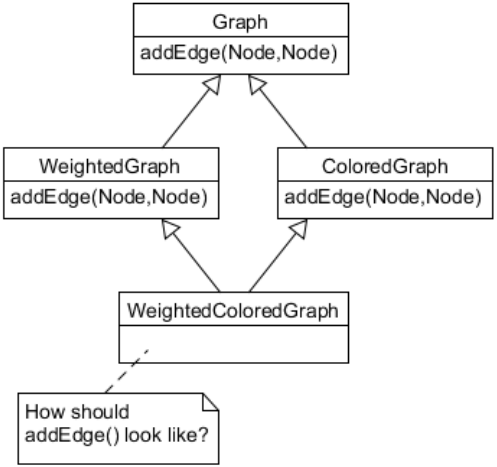
Feature Combinations?



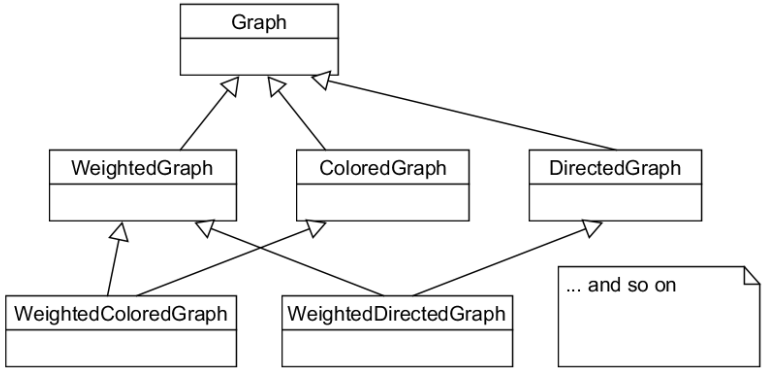
Diamond Problem

Multiple Inheritance

- most object-oriented programming languages do not support multiple inheritance (or only provide workarounds)
- critical: how to handle name clashes



Static Modeling of Feature Combinations

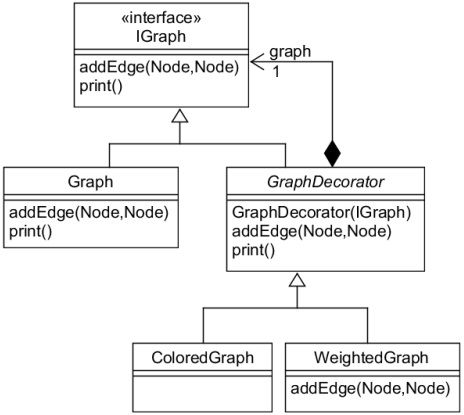


Even if multiple inheritance is supported, statically combining features through inheritance is tedious (or infeasible).

Decorator Pattern as a Solution?

```
abstract class GraphDecorator implements IGraph {
  IGraph graph;
  GraphDecorator(IGraph graph) {
    this.graph = graph;
  }
}
```

```
class WeightedGraph extends GraphDecorator {
  WeightedGraph(IGraph graph) {
    super(graph);
  }
  Edge add(Node n, Node m) {
    WeightedEdge e = (WeightedEdge) graph.add(n, m);
    e.weight = new Weight();
    return e;
  }
  ...
}
```



Example Usage

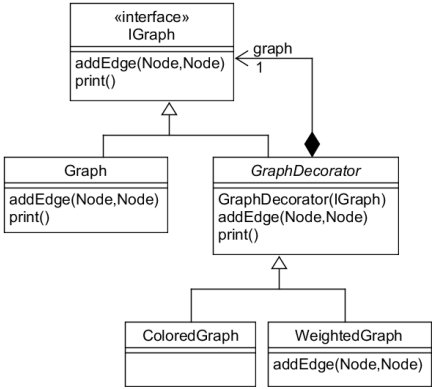
```
IGraph graph = new WeightedGraph(new ColoredGraph(new Graph(new WeightedEdgeFactory())));
```

Delegation Instead of Inheritance

Discussion

Extensions (i.e., features) can be combined dynamically, but ...

- must be independent of each other
- cannot add public methods
- runtime overhead due to indirections
- several physical objects are forming a conceptual one (e.g., problems with object identity)



Design Patterns for Variability – Summary

Lessons Learned

- Variability through object-orientation and design patterns
- Extension through delegation vs. inheritance
- Limitations and drawbacks w.r.t. feature combinations

Further Reading

- Meyer 1997, Chapter 3–4
- Gang of Four, Chapter 1–4

Practice

- What characterizes a modular software design and why can it support variability?
- In which sense are object-oriented solutions more modular than simple conditional statements?
- Do you know of other design patterns supporting variability?

FAQ – 2. Runtime Variability and Design Patterns

Lecture 2a

- What is variability, runtime variability, and binding time?
- What is the connection between variability, variability-intensive systems, and software product lines?
- How can configuration options be supplied?
- How can command-line parameters, preference dialogs, and configuration files be used to offer variability?
- What is the principle behind configuration options and when does the configuration happen?
- What are potential features of a graph library?
- What are examples for runtime variability? When are configuration options specified? By whom?
- When is the validity of configuration options checked?

Lecture 2b

- How to realize runtime variability in source code?
- What is the best strategy?
- How can (immutable) global variables and method parameters be used to realize variability?
- Why is the development of runtime variability challenging?
- What are code scattering, code tangling, and code replication?
- Does runtime variability allow reconfiguration at runtime?
- How to develop new features or variants with runtime variability? Exemplify!
- What are (dis)advantages of runtime variability?
- When (not) to use runtime variability?

Lecture 2c

- What are design patterns (used for)?
- Why are design patterns relevant for variable software?
- Which design patterns are relevant for variable software?
- What are intent, motivation, and idea for template method, abstract factory, and decorator pattern?
- Given an example class diagram, which pattern is applied or should be applied? Why?
- What is the diamond problem? How does it affect variable software?
- Is multiple inheritance useful to combine features? Exemplify!
- What are limitations of design patterns (for variable software)?