Recap

- Examples of *design patterns*
  - *Creational*: singleton, **builder**
  - *Structural*: composite, decorator
  - *Behavioral*: observer, strategy, state, **template method**
  - **Dependency Injection**

Objectives

- Dependency Injection and Template Method
- Software Metrics
  - Cyclomatic complexity
  - OO Software Metrics, Cohesion and Coupling

The Template Method Pattern

NAZARBAYEV UNIVERSITY
SCHOOL OF ENGINEERING AND DIGITAL SCIENCES

- Define the steps of an algorithm but allow the implementation details to be specified with subclasses
- Classes Involved:
  - Abstract Class
  - Concrete Class

Template Method[1]

"Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure."

[1] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

Examples

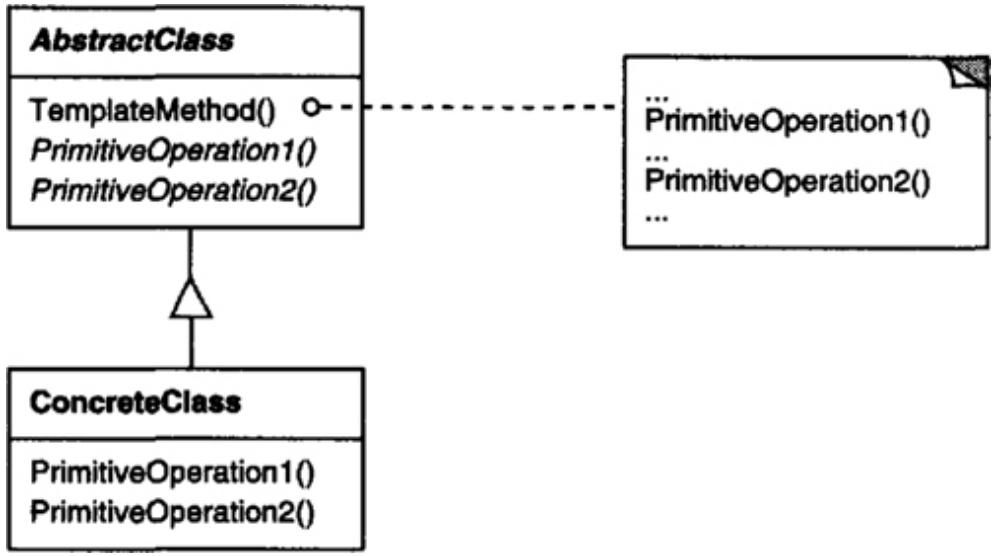
- The GoF example discusses a hypothetical example of a document editing framework
- A “template method” defines an algorithm in terms of abstract operations that subclasses can override
- In the example, the high-level abstraction might define a method for opening a document as a series of steps

Tea and Coffee Example[2]

- Tea and Coffee are similar in that both contain caffeine (usually) and are prepared using a similar sequence of steps
  - *boil the water*, *brew*, *add condiments*, ...
- Overall we would like to write the preparation of the beverage in some high-level abstraction (to avoid code duplication)
- Some steps of the preparation should be the same but some are specific to tea or coffee
  - Both begin with getting a kettle and boiling the water. However, we might want to add a different set of condiments for tea or coffee.

[2] Eric Freeman et al. *Head first design patterns*. O'Reilly Media, Inc., 2004.

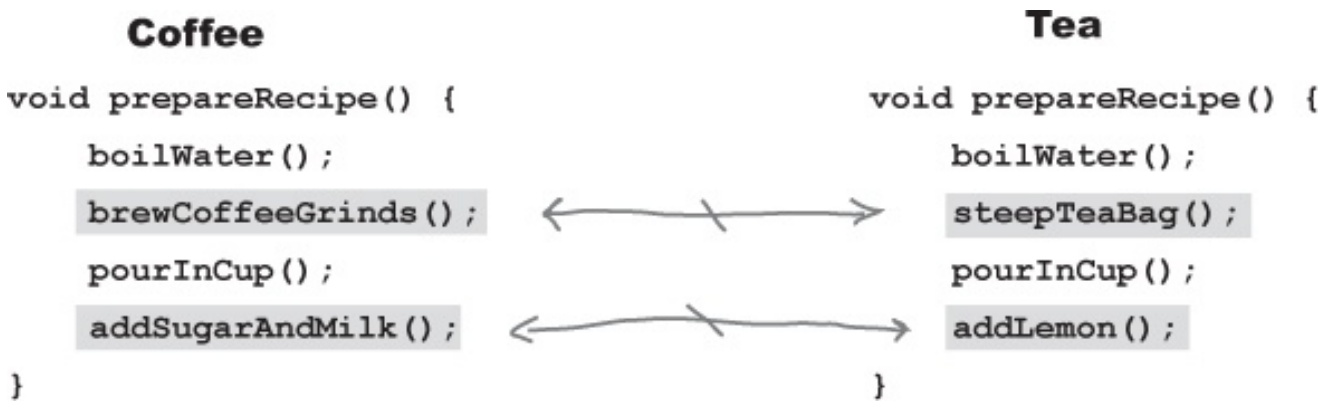
Template Method[3]



- The abstraction defines *primitive* operations that define the steps of an algorithm
- A *hook operation* provides default behavior that subclasses can extend
- Also has an inverted control structure (the so called *Hollywood Principle*)

[3]Class diagram from Gamma, *Design patterns: elements of reusable object-oriented software*, op. cit.

Coffee and Tea Example



- The diagram (image credit[4]) sketches the preparation for coffee and tea
  - Some things are shared and some things vary between the two
- `final` allows us to fix a method in the super class and `abstract` provides a method that *must* be implemented in subclasses

[4] Eric Freeman et al. *Head first design patterns*. O'Reilly Media, Inc., 2004.

Applications of Template Method

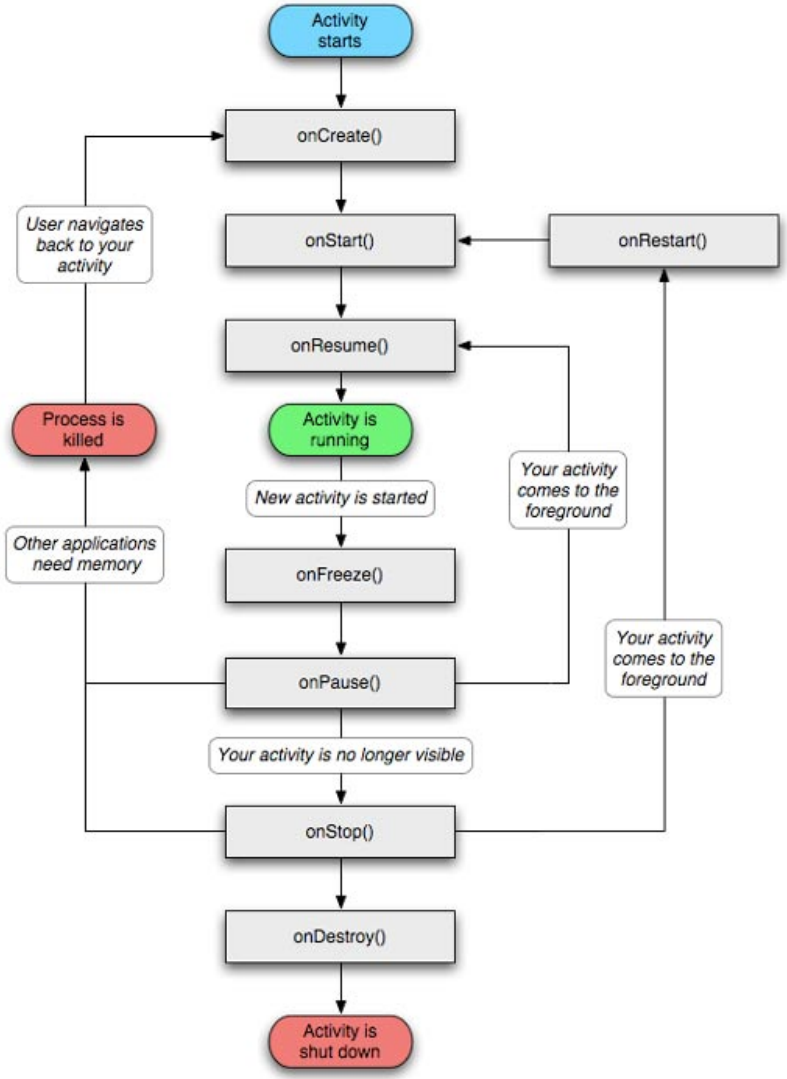
- According to Larman[5], Template Method is "at the heart of framework design".
- We often see the pattern in situations where software objects go through a life-cycle
  - Android development: the activity life-cycle
  - Java EE Servlets: Ch. 17 of the Java EE Tutorial[6]

[5] Craig Larman. *Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Interative Development*. Prentice Hall PTR, 2012.

[6] Eric Jendrock et al. *The Java EE Tutorial, Release 7*. Oracle, 2014.

Android Activity Lifecycle[7]

- The `Activity` class represents a single-screen in Android development, and is one of the main classes that devs work with
- The Activity lifecycle is shown at right
- Devs customize their activities by overriding the lifecycle methods, such as `onCreate()`



[7] `https://developer.android.com/guide/components/activities/activity-lifecycle`

Java Servlets

- Part of Java Enterprise Edition: extension of Java Standard Edition for server-side applications
- Basic `Servlet` class has three lifecycle methods init, service, and destroy
- Usually will use the package `javax.servlet.http` which contains methods to handle HTTP requests
- HTTP *Servlet*
  - Java class that can respond to HTTP requests
  - Override methods corresponding to different HTTP verbs

Simple Hello World Servlet


```
@WebServlet(urlPatterns={"/hello"})
public class HelloWorld extends HttpServlet
{
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                      throws ServletException, IOException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println(
         "<html><head><title>Hello Client!</title></head><body>"
       + "<h1>Hello Client!</h1>"
       + "</body></html>" );
    }
}
```

- Extended the base HTTP Servlet class and overrode the `doGet()` method

Open-Closed Principle (Template Method)

General Design Principle

Classes should be open for extension but closed for modification

- Want to extend functionality without having to modify source code in the base classes
- Previously we saw the Open-Closed Principle in the context of Decorator
- Template Method also illustrates this principle, we customize the behavior without modifying the base classes

Dependency Injection

NAZARBAYEV UNIVERSITY
SCHOOL OF ENGINEERING AND DIGITAL SCIENCES

```
public class MyClass {
    DependencyA a;
    DependencyB b;

    public MyClass(DependencyA a, DependencyB b) {
        this.a = a;
        this.b = b;
    }

}
```

- *Dependency Injection* refers to the way a class receives its dependencies from other parts of the application
- A dependency is simply some other class that is needed
- Dependencies passed in as constructor parameters: *constructor injection*
- Class not responsible for resolving its dependencies
- Related Ideas: Dependency-Inversion, Inversion of Control

Alternative Without DI

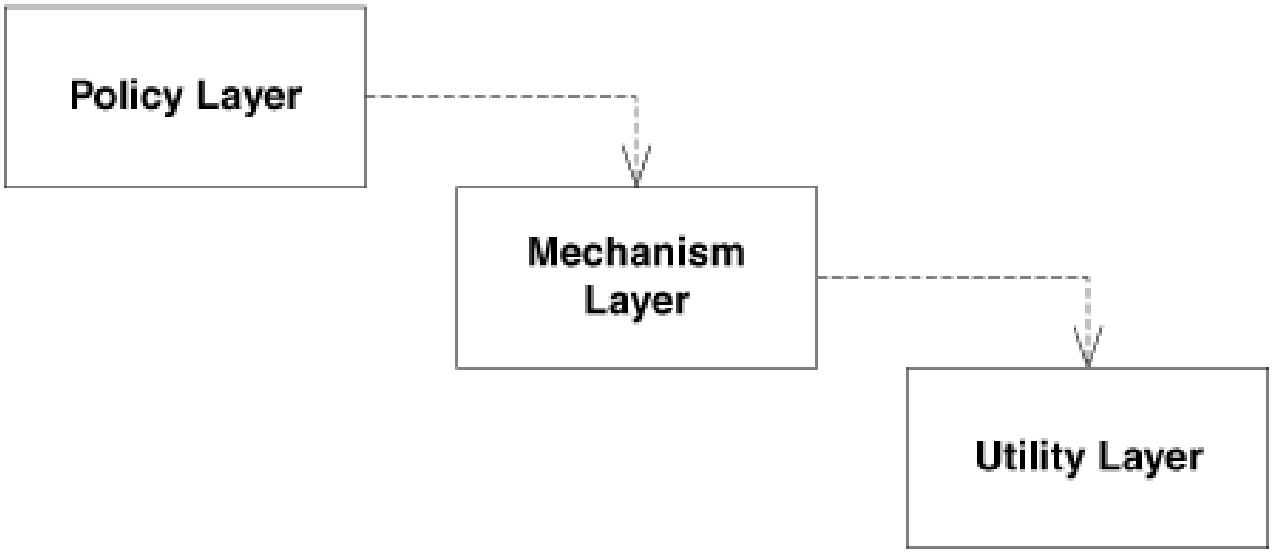

```
public class MyClass {
    DependencyA a = new RealDependencyA();
    DependencyB b = new RealDependencyB();
}
```

- Without using DI the class under development instantiates its dependencies using `new`
- Makes testing difficult
- Potential errors and hints not exposed in the signature

Dependency Injection Frameworks

- This pattern is now so widespread in software development that there are entire frameworks devoted to it
- Wiring up objects together is one of the most tedious parts of programming, so why not automate it
- Spring: provides an "inversion of control" container for Java
- Guice: light DI framework for Java developed by Google

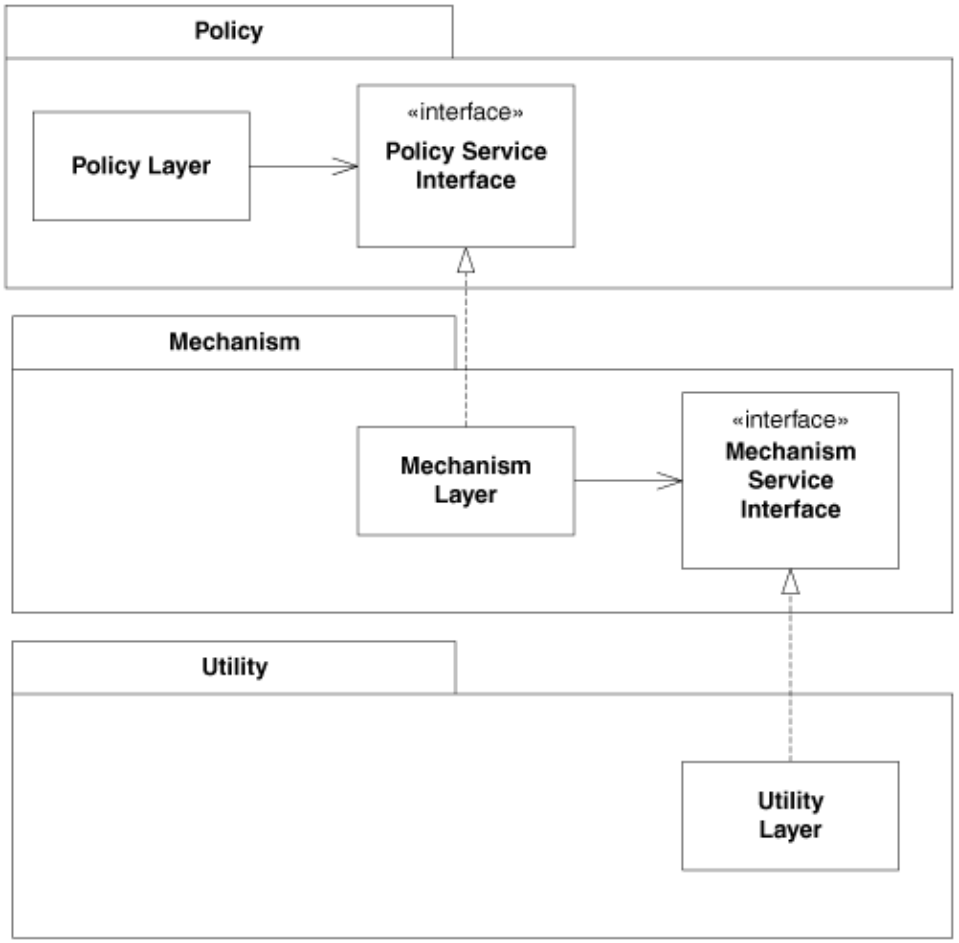
Dependency Inversion Principle (figure[8])



- Demonstrates how we might think about design in a procedural sense
- Begin at the top-layer and implement lower level details
- The top policy layer depends on implementation details all the way down to the utility layer

[8] Robert C. Martin. *Agile Principles, Patterns, and Practices in C#*. Prentice Hall, 2006.

Ownership Inversion (figure[9])



- Each higher level layer declares an abstract interface for the services it requires

[9] Robert C. Martin. *Agile Principles, Patterns, and Practices in C#*. Prentice Hall, 2006.

Ownership Inversion cont.

- In the OO case we invert our thinking with respect to dependencies
- Lower level components depend on upper level components
- Also applies to the ownership of interfaces
  - In this case, the upper layers declare interfaces that the lower layers derive from

Use Abstractions

- One approach to applying the dependency inversion principle in practice is to *depend on abstractions*
- References should be abstract classes or an interfaces
  - No variable should hold a reference to a concrete class.
  - No class should derive from a concrete class.
  - No method should override an implemented method of any of its base classes.

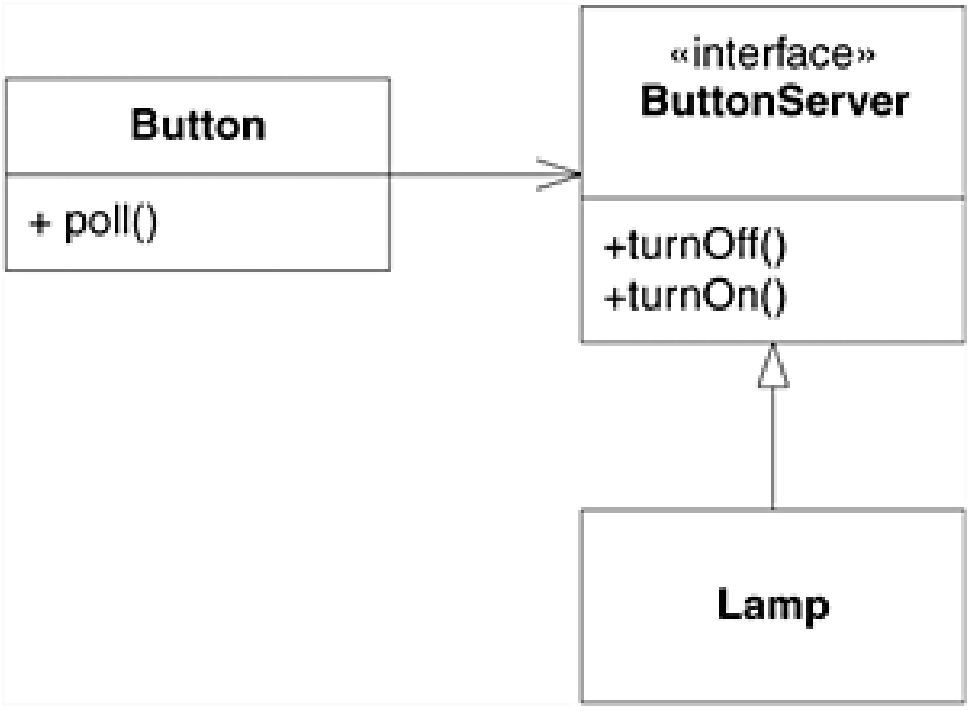
Simple Example[10]


```
public class Button
{
  private Lamp lamp;
  public void Poll()
  {
    if (/*some condition*/)
      lamp.TurnOn();
  }
}
```

- In this case we model the DI principle using the example of a button and a lamp
- The `Button` class depends directly on the `Lamp`

[10] Robert C. Martin. *Agile Principles, Patterns, and Practices in C#*. Prentice Hall, 2006.

Example w/Dependency Inversion (figure[11])



- In this case the policy contains a reference to an interface type
- The lamp derives from the interface
- The button can now control any device that implements the interface

[11] Robert C. Martin. *Agile Principles, Patterns, and Practices in C#*. Prentice Hall, 2006.

Designing for Change

- Most systems will need to respond to change at some point in their lifetime
- The underlying principle of the GoF patterns is designing for change and for the possibility of reuse in the future
  - This is achieved by letting some part of the system vary without upsetting the remainder (*encapsulate what varies*[12])
- *Example*: in the Strategy pattern we isolate variations in an algorithm by creating an abstract type and delegating
- People have created various mnemonics or lists for expressing principles of good oo design:

[12] Eric Freeman et al. *Head first design patterns*. O'Reilly Media, Inc., 2004.

Head First Principles

- Freeman *et al*[13] develop OO principles in the context of many of the GoF patterns
- **"Encapsulate what varies"**
- **"Favor composition over inheritance"**
- **"Strive for loosely coupled designs"**
- **Open-Closed Principle**: see *Decorator*
- **Depend on abstractions rather than concretions**

[13] Eric Freeman et al. *Head first design patterns*. O'Reilly Media, Inc., 2004.

Assigning Responsibilities

- A large part of Object-Oriented Design consists of assigning responsibilities to objects
- Using patterns allows us to apply "chunking" to this task
  - *Chunking* is the idea that it is easier for us to process information that has been grouped (i.e. chunked). *Example*: memory

GRASP Patterns[15]

- Larman[14] offers the GRASP patterns (some are listed below): "fundamental principles of object design and responsibility assignment"
- **Information Expert**: "assign a responsibility to the information expert"
- **High Cohesion**: "assign a responsibility so that cohesion remains high"
- **Low Coupling**: "assign a responsibility so that coupling remains low"

[14] Craig Larman. *Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Interative Development*. Prentice Hall PTR, 2012.

[15] General responsibility assignment software patterns

Use Software Metrics

- *Software Metrics* or *Code Metrics* measure things about software
  - Number of lines of code
  - Execution time
  - ...
- Metrics exist that quantify OO design principles and have been experimentally validated[16]
- Coupling: Coupling Between Object Classes (CBO), counts number of classes to which one class is coupled (use of member functions or instance variables)
- Cohesion: Lack of Cohesion on Methods (LCOM), measures how much the instance variables of a class are shared, compares number of *pairs* of member functions with and without shared instance variables

[16] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. "A validation of object-oriented design metrics as quality indicators". In: *IEEE Transactions on software engineering* 22.10 (1996), pp. 751–761.

Program Control Graph

NAZARBAYEV UNIVERSITY
SCHOOL OF ENGINEERING AND DIGITAL SCIENCES

- A very well known software metric is the *cyclomatic* or McCabe complexity[17]
- A *graph* is an abstract mathematical concept that consists of a set of nodes and a set of edges between them
- The *program control graph* models the different possible control flows in the program
  - A node corresponds to a block of code where execution is sequential
  - An edge corresponds to different branches taken in the program

[17] Thomas J McCabe. "A complexity measure". In: *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320.

Cyclomatic Complexity

$$v(G) = e - n + 2p$$

- The cyclomatic complexity of a program control graph is defined by the equation above
- Equal to the maximum number of linearly independent circuits
- The size of a basis of paths for the graph
- p is the number of connected components
- Usually we assume unique entry and exit points so that $p = 1$

Cyclomatic Complexity

$$v(G) = e - n + 2$$

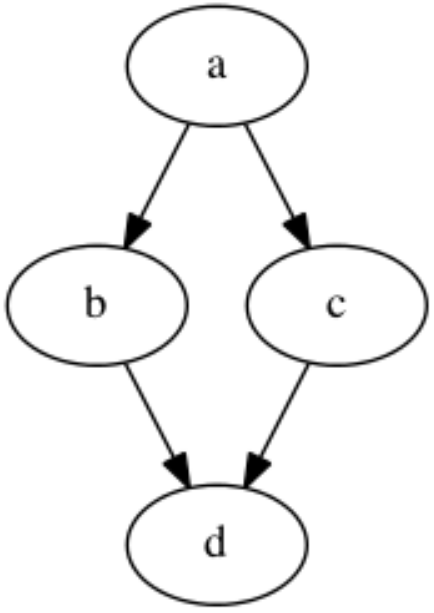
- Make the assumption that we have a single strongly connected component
- Also can be shown to count the overall number of branching statements in the program

Cyclomatic Complexity as a Measure of Program Complexity

NAZARBAYEV UNIVERSITY
SCHOOL OF ENGINEERING AND DIGITAL SCIENCES

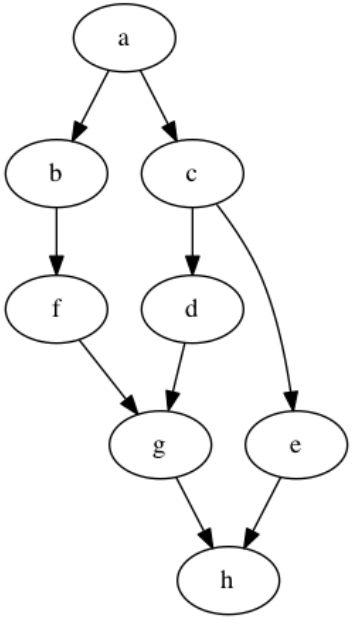
- The cyclomatic complexity is a measure of the "size" of the program, like the number of lines of code
- For a *planar* program control graph CC counts the number of distinct regions in the plane
- A program with a higher cyclomatic complexity is expected to require more effort to test

CC Example: simple IF



- This graph represents the program control flow for a simple `IF` statement.
- There are **4** nodes and **4** edges which yields a cyclomatic complexity of $v = e - n + 2 = 4 - 4 + 2 = 2$
  - Note that the plane is divided into 2 partitions by the flow graph

CC Example: more complex example



- In this case there are **8** nodes and **9** edges connected with more complicated branching
- $v = 9 - 8 + 2 = 3$
- As in the previous example, the CC complexity is equal to the number of planar regions created by the graph

Branch Coverage vs. Path Coverage

- *Code Coverage* is the degree to which a set of tests represents the possible behavior of a program
  - Branch Coverage: each branch of a control structure is executed
  - Path Coverage: every possible execution route has been taken
- Clearly, Branches $\leq$ Paths. We do not expect to achieve path coverage on moderately complex programs
- Cyclomatic Complexity is an upper bound on the number of tests we would need to run for branch coverage

Object-Oriented Metrics

NAZARBAYEV UNIVERSITY
SCHOOL OF ENGINEERING AND DIGITAL SCIENCES

- Standard set of OO Metrics defined by Chidamber and Kemerer[18]
- Mention 3 in particular
  - Depth of Inheritance Tree (DIT)
  - Coupling Between Object Classes (CBO)
  - Lack of Cohesion on Methods (LCOM)
- Experimentally validaded by Basili *et al*[19]

[18] Shyam R Chidamber and Chris F Kemerer. "A metrics suite for object oriented design". In: *IEEE Transactions on software engineering* 20.6 (1994), pp. 476–493.

[19] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. "A validation of object-oriented design metrics as quality indicators". In: *IEEE Transactions on software engineering* 22.10 (1996), pp. 751–761.

Definition of LCOM[20]

The number of pairs of member functions without shared instance variables, minus the number of pairs of member functions with shared instance variables. However, the metric is set to zero whenever the above subtraction is negative.

- Number of pairs of methods is a binomial coefficient $\binom{N}{2}$ where N is the number of methods
- Higher LCOM $\rightarrow$ less cohesion, class has multiple responsibilities

[20] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. "A validation of object-oriented design metrics as quality indicators". In: *IEEE Transactions on software engineering* 22.10 (1996), pp. 751–761.

LCOM Example

NAZARBAYEV UNIVERSITY
SCHOOL OF ENGINEERING AND DIGITAL SCIENCES

```
public class Person {
  String name;
  String address;
  int age;

  public String getName() { return name.toUpperCase(); }
  public int getAge() { return age; }
  public String getDescription() {
    return getName() + " " + address;
  }
}
```

- `getName()` and `getAge()`: do not access any of the same instance variables
- `getAge()` and `getDescription()`: do not access any of the same instance variables
- `getName()` and `getDescription()`: both depend on `name`
- There are 2 pairs without shared instance variables and 1 pair with shared instance variables, LCOM equals $2 - 1 = \mathbf{1}$
- *Note*: we may choose to exclude getter and setter methods from the definition of LCOM

Why care about software metrics?

- Time available for testing is a finite resource
  - Know where to focus our resources
  - Estimate the amount of testing we need to do
- By being quantitative, metrics can help to establish *evidence* in Software Engineering
  - As in, something stronger than anecdote
- You can experiment with CC (in Javascript) at `jshint.com`

Reading

NAZARBAYEV UNIVERSITY
SCHOOL OF ENGINEERING AND DIGITAL SCIENCES

- Readings
  - Sections 2.3 and 2.4 of the lecture notes
  - McCabe[21] and Basili *et al*[22]

[21] Thomas J McCabe. "A complexity measure". In: *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320.

[22] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. "A validation of object-oriented design metrics as quality indicators". In: *IEEE Transactions on software engineering* 22.10 (1996), pp. 751–761.