Design Patterns (1): Creational Patterns

October 27, 2022

Byung-Gon Chun

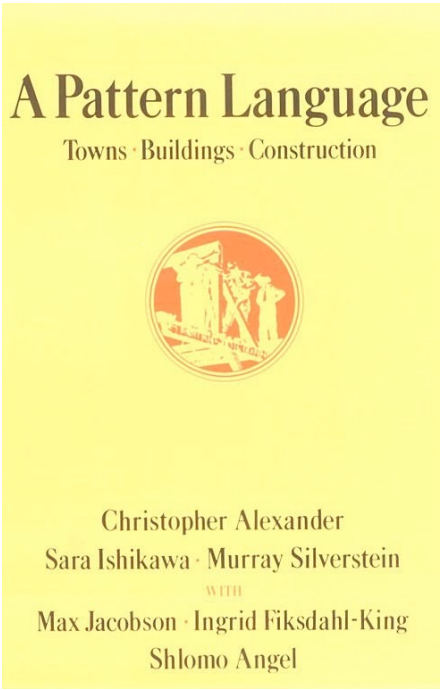
(Slide credits: George Candea, EPFL and Armando Fox, UCB)

Emerging Patterns

- “People should design for themselves their own houses, streets and communities. This idea comes simply from the observation that most of the wonderful places of the world were not made by architects but by the people”

Christopher Alexander (1977)

Emerging Patterns



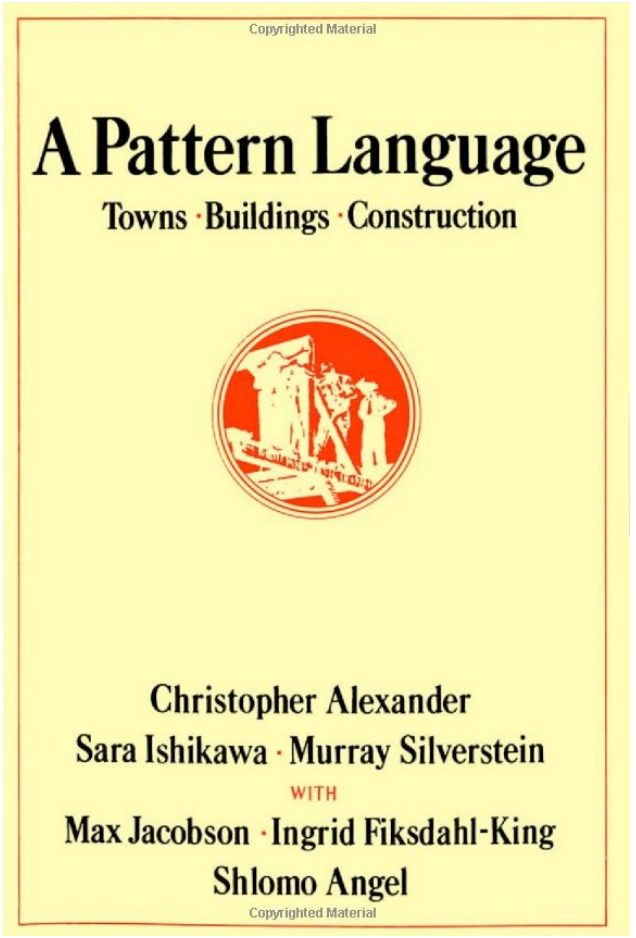
Christopher Alexander (1977)

- How to enable people to “build” ?
- How to arrange the physical environment to solve a problem?
- Examples: arcades, alcoves, cascade of roofs, garden wall, window place, filtered light, etc.

Design Patterns Promote Reuse

"A pattern describes a problem that occurs often, along with a tried solution to the problem" - Christopher Alexander, 1977

- Christopher Alexander's 253 (civil) architectural patterns range from the creation of cities (2. distribution of towns) to particular building problems (232. roof cap)
- A pattern language is an organized way of tackling an architectural problem using patterns
- *Separate the things that change from those that stay the same*



Challenges in Software Engineering

- Designing good object-oriented software
  - Pertinent classes? Interfaces? Inheritance? Relationship?
  - Experienced designers can get it right
  - Inexperienced ones spend lots of time and make mistakes

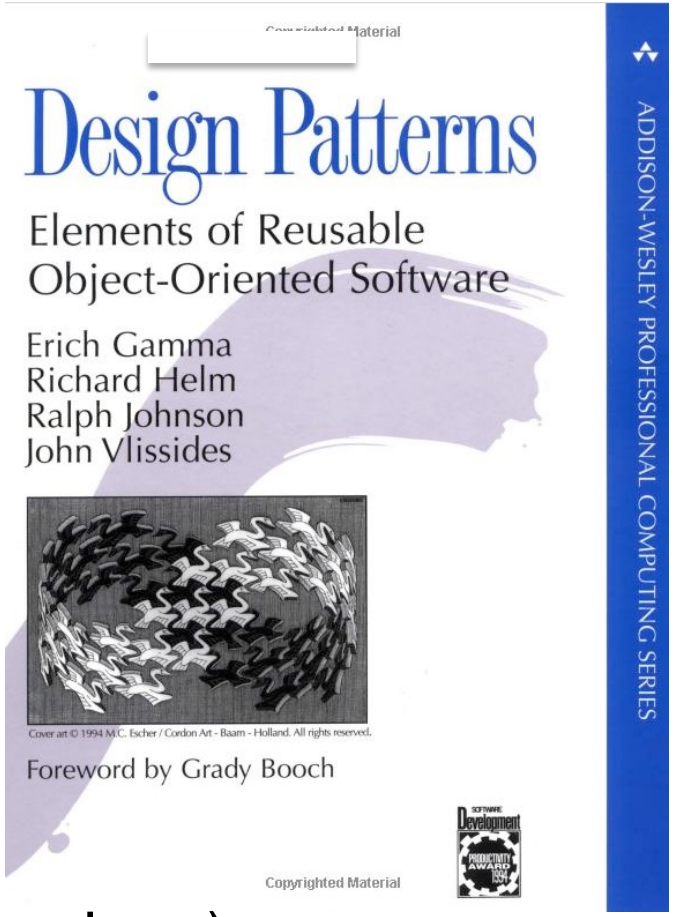
- Experience = toolbox of reusable solutions
  - Classify problems and apply solution templates

Kinds of Patterns in Software

- Architectural ("macroscale") patterns
  - Model-view-controller
  - Pipe & Filter (e.g. compiler, Unix pipeline)
  - Event-based (e.g. interactive game)
  - Layering (e.g. SaaS technology stack)
  - Dataflow (e.g., Map-Reduce)
- Computation patterns
  - Fast Fourier transform
  - Structured & unstructured grids
  - Dense linear algebra
  - Sparse linear algebra
- *Software design patterns*
  - *GoF (Gang of Four) Patterns: structural, creational, behavior*

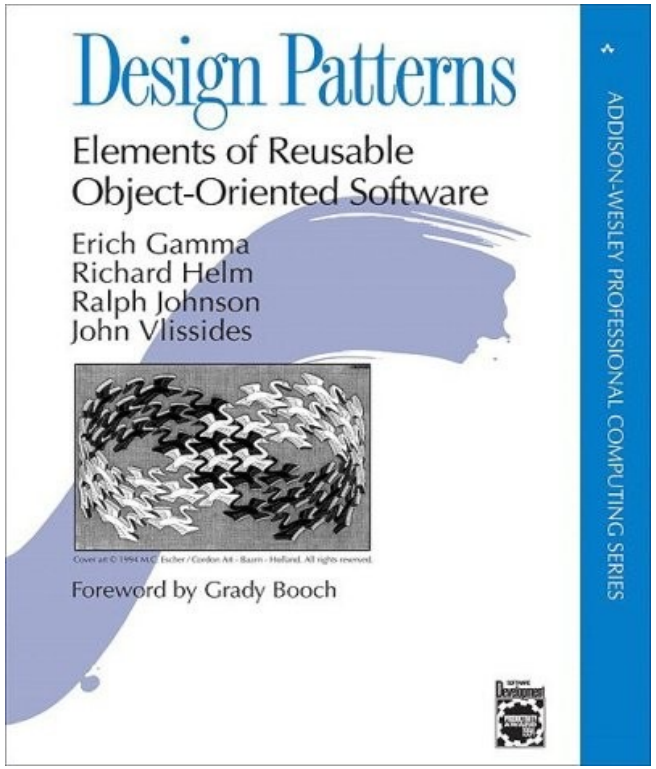
The Gang of Four (GoF)

- 23 design patterns
- description of communicating objects & classes
  - captures common (and successful) solution to a *category* of related problem instances
  - can be customized to solve a specific (new) problem in that category
- Pattern ≠
  - individual classes or libraries (list, hash, ...)
  - full design



Software Design Patterns

- Name
  - Establish a vocabulary
- Problem
  - When to apply it: problem definition + context
- Solution
  - Template = elements + responsibilities + relationships
- Consequences
  - Trade-offs (e.g., space vs. time)



Creational Patterns

Abstract Factory
Builder
Factory
Prototype
Singleton

Structural Patterns

Adaptor
Bridge
Composite
Decorator
Façade
Flyweight
Proxy

Behavioral Patterns

Chain of Responsibility
Command
Interpreter
Iterator
Mediator
Memento
Observer
State
Strategy
Template Method
Visitor

Architectural : Model-View-Controller
Service-oriented Architecture

Concurrency Patterns : Actor
Monitor
Thread Pool

Principles of Good Object-Oriented Design that Inform Patterns

Separate out the things that change from those that stay the same

Two overarching principles cited by the GoF authors

1. Program to an Interface, not an Implementation
2. Prefer Composition and Delegation over Inheritance

Antipattern

- Code that looks like it should probably follow some design pattern, but doesn't
- Often result of accumulated *technical debt*
- Symptoms:
  - Viscosity (easier to do hack than Right Thing)
  - Immobility (can't DRY out functionality)
  - Needless repetition (comes from immobility)
  - Needless complexity from generality

Basic Design Patterns

Basic Design Patterns

- Encapsulation
- Inheritance
- Iteration
- Exceptions

Encapsulation

- Problem = exposure can lead to
  - Violation of representation invariant
  - Dependencies that hamper implementation changes
- Solution = hide components
- Consequences
  - Interface may not provide all desired operations
  - Indirection may reduce performance

Inheritance

- Problem = similar abstractions…
  - Have similar fields and methods
  - Repeating them => tedious, error-prone, unmaintainable
- Solution = inherit default members
  - Correct implementation selected via runtime dispatching
- Consequences
  - Code for a subclass not contained all in one place
  - Runtime dispatching introduces overhead

Iteration

- Problem = accessing all collection members…
  - Requires specialized traversal
  - Exposes underlying details
- Solution = implementation does traversal
  - Results returned via standard interface
- Consequences
  - Iteration order constrained by implementation

Exceptions

- Problem = errors occur in one place…
  - But should be handled in another part of the code
  - Shouldn’t clutter code with error recovery
  - Shouldn’t mix return values with error codes
- Solution = specialized language structure
  - Throw exception in one place, catch & handle in another
- Consequences
  - Hard to know layer at which exception will be handled
  - Evil temptation to use this for normal control flow

Next

- Factory
- AbstractFactory
- Builder
- Dependency injection

Factory Method

Two “factory methods”

- *Static* Factory Method
  - *gain control over object creation*
- Original GOF (“gang of four”) definition
  - *delegate to subclasses the decision of what instance of a class to create*

Example

- Worker
  - *e.g., an instance of a thread that perform some work on behalf of requestor*

- Two constraints
  - *creation/deletion of worker may be expensive relative to the work it does*
  - *need to limit the number of workers, to preserve system performance*

```
public class Worker {
   // ...
   private static Set<Worker> availableWorkers = new HashSet<Worker>();
   private Worker() {
      // ... create a worker ...
      numWorkers++;
   }

   public static Worker getWorker() {
      if (numWorkers < MAX_WORKERS) {
         return new Worker();
      }
      else if (availableWorkers.size() > 0) {
         Worker worker = availableWorkers.iterator().next();
         availableWorkers.remove(worker);
         return worker;
      } else {
         throw new NoWorkersAvailable();
      }
   }

   public static void yieldWorker (Worker worker) {
      //...
   }
}
```

```
class Complex {
    public static Complex fromCartesian (double real, double imaginary) {
        return new Complex (real, imaginary);
    }

    public static Complex fromPolar (double modulus, double angle) {
        return new Complex (modulus * cos(angle), modulus * sin(angle));
    }

    private Complex (double a, double b) {
        //...
    }
}
```

Original GOF Factory Method

- Goal
  - *delegate to a factory method the decision of what instance of a class to create*
- Mechanics
  - *base class provides a method meant to be overriden by subclasses*
  - *subclasses decide what type of object to return*
    => this pattern is also called the “virtual constructor”

```java
public interface Printer {
   public void debug (String message); // write out a debug message
   public void error (String message); // write out an error message
}

public class ConsolePrinter implements Printer {
   // ...
}

class Worker {
  public void doWork() {
     Printer log = new ConsolePrinter();
     // ...
     log.debug ("Sending POST to server");
     // ...
     if (response != SUCCESS) {
        log.error ("Received error from server");
     }
     // ...
  }
}
```

How to support another Printer implementation?

```java
public class FilePrinter implements Printer {
   // ...
}
```

```
public interface Printer {
   public void debug (String message); // write out a debug message
   public void error (String message); // write out an error message
}

public class ConsolePrinter implements Printer {
   // ...
}

class Worker {
  protected Printer getPrinter () {
    return new ConsolePrinter();
  }
  public void doWork() {
     Printer log = getPrinter();
     // ...
     log.debug ("Sending POST to server");
     // ...
     if (response != SUCCESS) {
        log.error ("Received error from server");
     }
     // ...
  }
}
```

```
public class FilePrinter implements Printer {
   // ...
}

class WorkerWithFileLogging extends Worker {
  Printer getPrinter() {
    return new FilePrinter();
  }
}
```

*Disclaimer: not the best way to tackle this problem

Python Examples

```
from django import forms

class PersonForm(forms.Form):
   name = forms.CharField(max_length=100)
   birth_date = forms.DateField(required=False)
```

Two “Factory Methods”

- Original GOF definition
  - *base class provides interface for producing objects, subclasses produce them*
    => polymorphism is a key ingredient of Factory Method
  - *the Factory Method is always non-static (to be overrideable)*
- *Static* Factory Method
  - *gain control over object creation => can implement pools of objects, control amount of resources, ...*
  - *Note. subclasses cannot access any constructors of its super class*

Abstract Factory

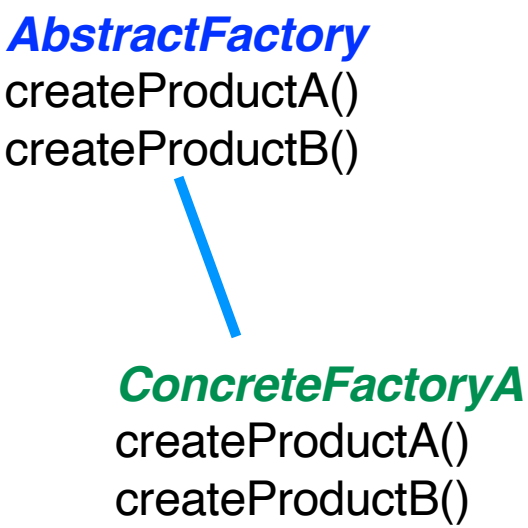
Problem

- Assemble something without regard to component details
  - *e.g., different families of components, yet keep the assembly process generic*
- Want to delegate the enforcement of consistency of use
  - *make sure the product family's rules of use are enforced, but not have the logic for doing so in the client code*

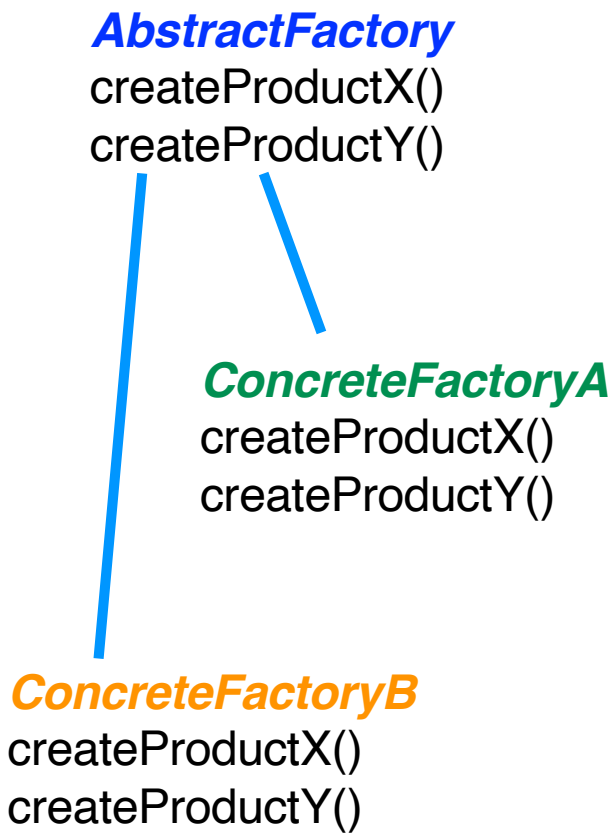
Solution Template

ConcreteFactoryA
createProductA()
createProductB()

Solution Template

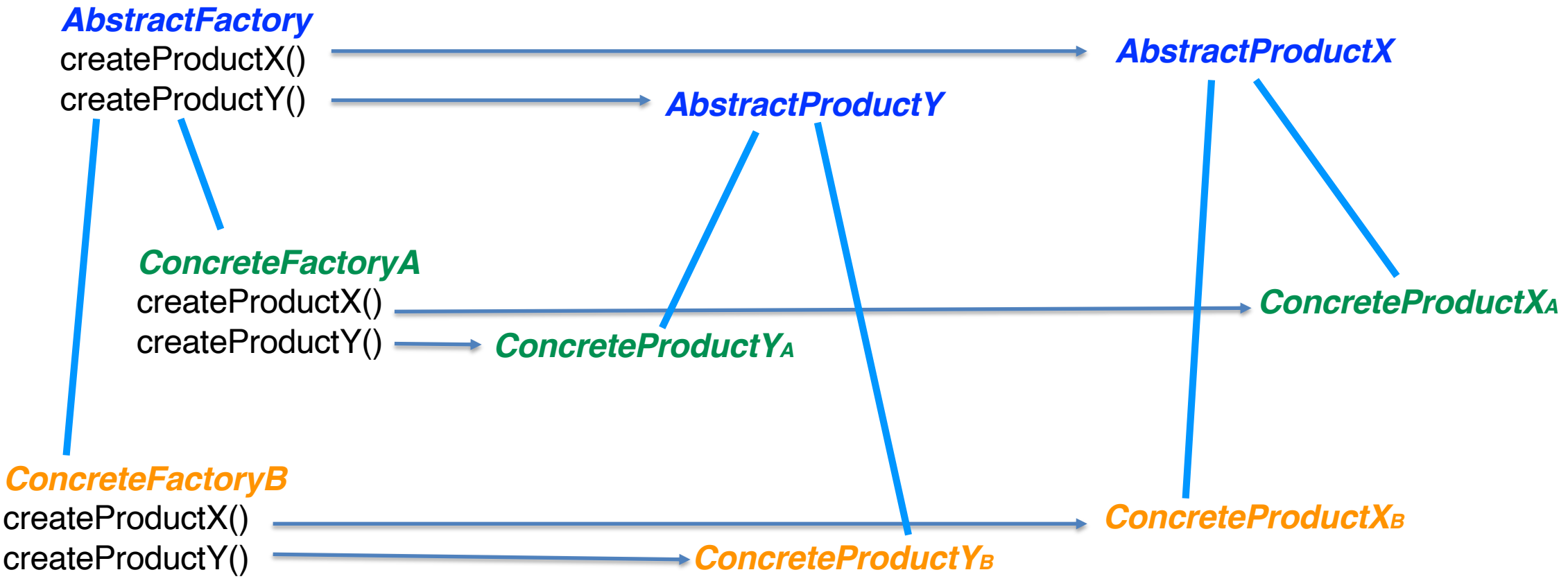


Solution Template



Solution Template

Solution Template



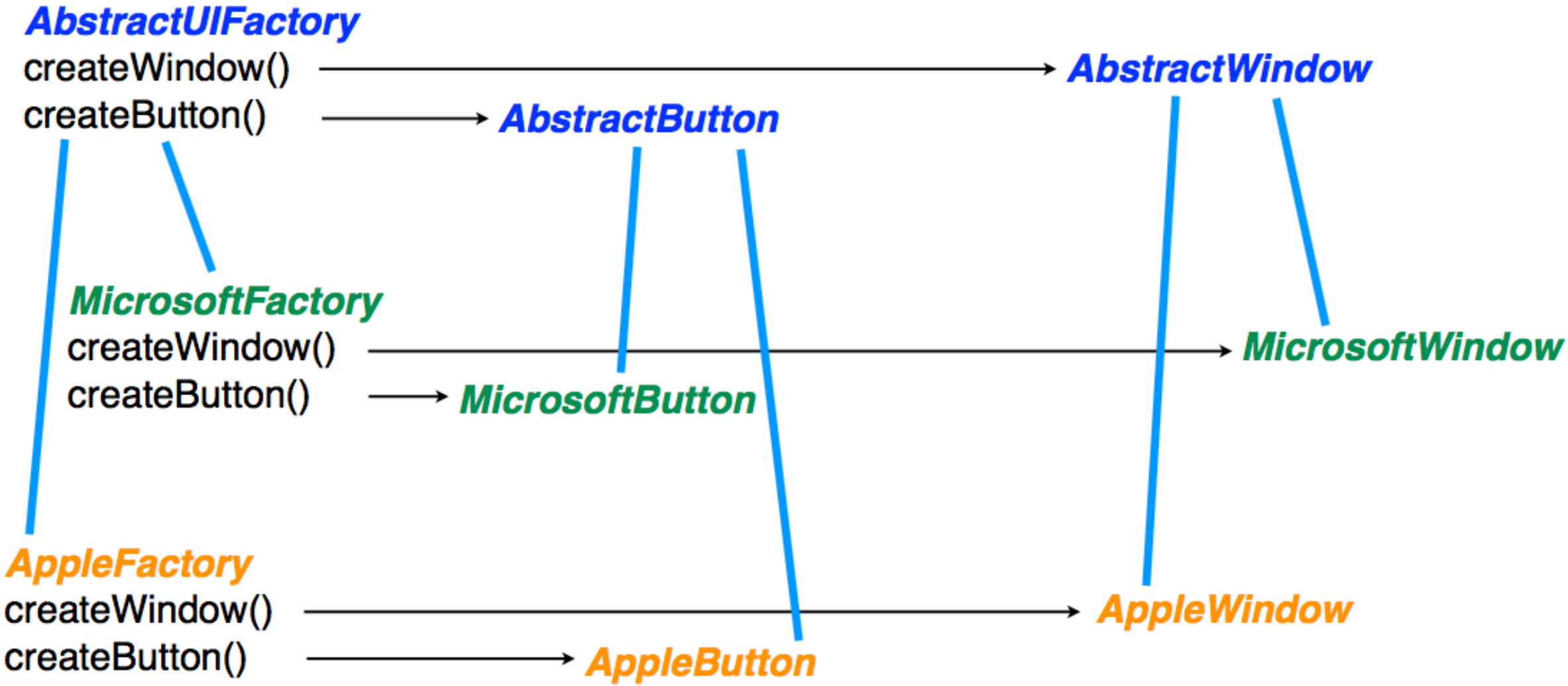
```
// ...
public Application (UIFactory factory) {
    Window window = factory.createWindow();
    Button buttonCancel = factory.createButton (window, "Cancel");
    Button buttonOk = factory.createButton (window, "OK");
    window.display();
    buttonCancel.activate();
    buttonOk.activate();
    // ...
}
// ...

Application app = new Application( /* factory corresponding to desired look & feel */ );
```

```
interface UIFactory {
    // ...
    public Window createWindow();
    public Button createButton();
    // ...
}

interface Window {
    // ...
    public void display();
}

interface Button {
    // ...
    public void activate();
}
```

```
class MicrosoftFactory implements UIFactory {

    // ...

    public Window createWindow() {
        return new MicrosoftWindow();
    }

    public Button createButton( Window) {
        return new MicrosoftButton();
    }
}

class MicrosoftWindow implements Window {
    // ...
    public void display() {
        // ...
    }
}

class MicrosoftButton implements Button {
    // ...
    public void activate() {
        // ...
    }
}
```

```
class AppleFactory implements UIFactory {

    // ...

    public Window createWindow() {
        return new AppleWindow();
    }

    public Button createButton( Window) {
        return new AppleButton();
    }
}

class AppleWindow implements Window {
    // ...
    public void display() {
        // ...
    }
}

class AppleButton implements Button {
    // ...
    public void activate() {
        // ...
    }
}
```

SolutionTemplate



Consequences

- Benefits
  - Client code uses multiple families of products, yet is isolated from their implementation
  - Switch between product families with minimal code disruption
  - Enforce inter-product constraints uniformly in the concrete factories
- Liabilities
  - Hard to add/remove products or product attributes
  - Potentially unnecessary complexity and extra work writing the initial code

Builder

Problem

- An object must be created in multiple steps
- Different representations of the same construction are required
- Telescopic constructor problem in Java – forced to create a new constructor for supporting different ways of creating an object

Factory vs. Builder

- A factory pattern creates an object in a single step, whereas a builder pattern creates an object in multiple steps

- E.g., protobuf Message Builder

Java

Two optional parameters

```
Socket()
Socket(String host)
Socket(int port)
Socket(String host, int port)
```

Add one parameter boolean ssl

```
Socket()
Socket(String host)
Socket(int port)
Socket(String host, int port)
Socket(boolean ssl)
Socket(String host, boolean ssl)
Socket(int port, boolean ssl)
Socket(String host, int port, boolean ssl)
```

Inner static class Builder

```
public class Socket {
    private final int port;
    private final String host;
    private final boolean ssl;
    private final int timeout;

    private Socket(Builder builder) {
        this.port = builder.port;
        this.host = builder.host;
        this.ssl = builder.ssl;
        this.timeout = builder.timeout;
    }

    public static class Builder {
        private int port = 0;
        private String host = null;
        private boolean ssl = false;
        private int timeout = 0;

        public Builder port(int port) {
            this.port = port;
            return this;
        }

        public Builder host(String host) {
            this.host = host;
            return this;
        }

        public Builder ssl(boolean ssl) {
            this.ssl = ssl;
            return this;
        }

        public Builder timeout(int timeout) {
            this.timeout = timeout;
            return this;
        }

        public Socket build() {
            return new Socket(this);
        }
    }
}
```

```
Socket s1 = new Socket.Builder().port(8080).timeout(60).build();

Socket s2 = new Socket.Builder().ssl(true).build();
```

Singleton

Problem

- There shall only be one object
- Need a well known broker for a resource
  - *i.e., point that controls access to a shared resource must be known*
- Resource accessed from disparate parts of the system
  - *they don't know about each other, so it's harder for them to coordinate*

Solution Template

- Single instance of a class
- Have the class enforce uniqueness

```
// Singleton with public final field
public class Singleton {

    public static final Singleton oneInstance = new Singleton();

    private Singleton() {}

}
```

```java
// Singleton with static factory
public class Singleton {

    private static final Singleton oneInstance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return oneInstance;
    }
}
```

```java
// Singleton with lazy initialization
public class Singleton {

    private static Singleton oneInstance = null;

    private Singleton() {}

    public static Singleton getInstance() {
        if (oneInstance == null) {
            synchronized (Singleton.class) {
                if (oneInstance == null) {
                    oneInstance = new Singleton();
                }
            }
        }
        return oneInstance;
    }
}
```

```
// Singleton with static factory,
// defending against reflection attack
public class ReflectionAttack {
    Public static void main(String[] args) throws Exception {
        Singleton singleton = Singleton.oneInstance;
        Constructor constructor = singleton.getClass().getDeclaredConstructor(new Class[0]);
        constructor.setAccessible(true);

        Singleton singleton2 = (Singleton) constructor.newInstance();
    }
}
```

```
// Singleton with static factory,
// defending against reflection attack
public class Singleton {
    private static final Singleton oneInstance = new Singleton();
    private Singleton() {
        if (oneInstance != null) {
            throw new IllegalStateException("Already instantiated");
        }
    }

    public static Singleton getInstance() {
        return oneInstance;
    }
}
```

```java
// Singleton (simplest and safest from in Java 5 and above)
public enum Singleton {
    oneInstance;
}
```

```java
// Singleton (simplest and safest from in Java 5 and above)
public enum Singleton {
    oneInstance;

    int value;

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }
}
```

Consequences

- Good
  - *can enforce strict control over access to the resource*
  - *better than a global variable or mutable public field*

- Bad
  - *makes testing harder! Global variable in disguise!*
  - *hard to subclass*
  - *hard to keep unique in Java*

Dependency Injection

Dependency Injection

- Make code modular and testable
- Make wiring everything together easy

Writing billing code for a pizza ordering website (https://github.com/google/guice)

```
public interface BillingService {
  /**
   * Attempts to charge the order to the credit card. Both successful and
   * failed transactions will be recorded.
   *
   * @return a receipt of the transaction. If the charge was successful, the
   * receipt will be successful. Otherwise, the receipt will contain a
   * decline note describing why the charge failed.
   */
  Receipt chargeOrder(PizzaOrder order, CreditCard creditCard);
}
```

Along with the implementation, we'll write unit tests for our code.

Direct constructor calls

```
public class RealBillingService implements BillingService {
  public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {
    CreditCardProcessor processor = new PaypalCreditCardProcessor();
    TransactionLog transactionLog = new DatabaseTransactionLog();
    try {
      ChargeResult result = processor.charge(creditCard, order.getAmount());
      transactionLog.logChargeResult(result);

      return result.wasSuccessful()
          ? Receipt.forSuccessfulCharge(order.getAmount())
          : Receipt.forDeclinedCharge(result.getDeclineMessage());
    } catch (UnreachableException e) {
      transactionLog.logConnectException(e);
      return Receipt.forSystemFailure(e.getMessage());
    }
  }
}
```

Factories

- A factory class decouples the client and implementing class.

- A simple factory uses static methods
  - to get implementations for interfaces.

  - to pass implementations for interfaces in the constructor

  or to set implementations for interfaces

```
public class CreditCardProcessorFactory {
  private static CreditCardProcessor instance;

  public static void setInstance(CreditCardProcessor processor) {
    instance = processor;
  }

  public static CreditCardProcessor getInstance() {
    if (instance == null) {
      return new SquareCreditCardProcessor();
    }
    return instance;
  }
}
```

```
public class RealBillingService implements BillingService {
  public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {
    CreditCardProcessor processor = new PaypalCreditCardProcessor();
    TransactionLog transactionLog = new DatabaseTransactionLog();
    try {
      ChargeResult result = processor.charge(creditCard, order.getAmount());
      transactionLog.logChargeResult(result);

      return result.wasSuccessful()
          ? Receipt.forSuccessfulCharge(order.getAmount())
          : Receipt.forDeclinedCharge(result.getDeclineMessage());
    } catch (UnreachableException e) {
      transactionLog.logConnectException(e);
      return Receipt.forSystemFailure(e.getMessage());
    }
  }
}
```

```
public class RealBillingService implements BillingService {
  public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {
    CreditCardProcessor processor = CreditCardProcessorFactory.getInstance();
    TransactionLog transactionLog = TransactionLogFactory.getInstance();
    try {
      ChargeResult result = processor.charge(creditCard, order.getAmount());
      transactionLog.logChargeResult(result);

      return result.wasSuccessful()
          ? Receipt.forSuccessfulCharge(order.getAmount())
          : Receipt.forDeclinedCharge(result.getDeclineMessage());
    } catch (UnreachableException e) {
      transactionLog.logConnectException(e);
      return Receipt.forSystemFailure(e.getMessage());
    }
  }
}
```

```
public class RealBillingServiceTest extends TestCase {
  private final PizzaOrder order = new PizzaOrder(100);
  private final CreditCard creditCard = new CreditCard("1234", 11, 2010);

  private final InMemoryTransactionLog transactionLog = new InMemoryTransactionLog();
  private final FakeCreditCardProcessor processor = new FakeCreditCardProcessor();

  @Override
  public void setUp() {
    TransactionLogFactory.setInstance(transactionLog);
    CreditCardProcessorFactory.setInstance(processor);
  }
  @Override
  public void tearDown() {
    TransactionLogFactory.setInstance(null);
    CreditCardProcessorFactory.setInstance(null);
  }
  public void testSuccessfulCharge() {
    RealBillingService billingService = new RealBillingService();
    Receipt receipt = billingService.chargeOrder(order, creditCard);

    assertTrue(receipt.hasSuccessfulCharge());
    assertEquals(100, receipt.getAmountOfCharge());
    assertEquals(creditCard, processor.getCardOfOnlyCharge());
    assertEquals(100, processor.getAmountOfOnlyCharge());
    assertTrue(transactionLog.wasSuccessLogged());
  }
}
```

Problems with Factory-based Code

- A global variable holds the mock implementation, so we need to be careful about setting it up and tearing it down
- Should the tearDown fail, the global variable continues to point at our test instance. This could cause problems for other tests. It also prevents us from running multiple tests in parallel.
- The biggest problem is that the dependencies are *hidden in the code*
  - If we add a dependency on a CreditCardFraudTracker, we have to re-run the tests to find out which ones will break.
  - Should we forget to initialize a factory for a production service, we don't find out until a charge is attempted.
  - As the application grows, babysitting factories becomes a growing drain on productivity.

Dependency Injection

- Like the factory, dependency injection is just a design pattern. The core principle is to *separate behavior from dependency resolution*.

- The RealBillingService is not responsible for looking up the TransactionLog and CreditCardProcessor. Instead, they're passed in as constructor parameters:

```
public class RealBillingService implements BillingService {
  private final CreditCardProcessor processor;
  private final TransactionLog transactionLog;

  public RealBillingService(CreditCardProcessor processor, TransactionLog transactionLog) {
    this.processor = processor;
    this.transactionLog = transactionLog;
  }

  public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {
    try {
      ChargeResult result = processor.charge(creditCard, order.getAmount());
      transactionLog.logChargeResult(result);

      return result.wasSuccessful()
          ? Receipt.forSuccessfulCharge(order.getAmount())
          : Receipt.forDeclinedCharge(result.getDeclineMessage());
    } catch (UnreachableException e) {
      transactionLog.logConnectException(e);
      return Receipt.forSystemFailure(e.getMessage());
    }
  }
}
```

Test

```
public class RealBillingServiceTest extends TestCase {

  private final PizzaOrder order = new PizzaOrder(100);
  private final CreditCard creditCard = new CreditCard("1234", 11, 2010);

  private final InMemoryTransactionLog transactionLog = new InMemoryTransactionLog();
  private final FakeCreditCardProcessor processor = new FakeCreditCardProcessor();

  public void testSuccessfulCharge() {
    RealBillingService billingService = new RealBillingService(processor, transactionLog);
    Receipt receipt = billingService.chargeOrder(order, creditCard);

    assertTrue(receipt.hasSuccessfulCharge()); assertEquals(100, receipt.getAmountOfCharge());
    assertEquals(creditCard, processor.getCardOfOnlyCharge());
    assertEquals(100, processor.getAmountOfOnlyCharge());
    assertTrue(transactionLog.wasSuccessLogged());
  }
}
```

The dependency is *exposed in the API signature*

Whenever we add or remove dependencies, the compiler will remind us what tests need to be fixed.

- Now the clients of BillingService need to lookup its dependencies.
- We can fix some of these by applying the pattern again! Classes that depend on it can accept a BillingService in their constructor.
- For top-level classes, it's useful to have a framework. Otherwise you'll need to construct dependencies recursively when you need to use a service

```
public static void main(String[] args) {
  CreditCardProcessor processor = new PaypalCreditCardProcessor();
  TransactionLog transactionLog = new DatabaseTransactionLog();
  BillingService billingService = new RealBillingService(processor, transactionLog);
  ...
}
```

Inversion of Control

- In traditional programming, the custom code that expresses the purpose of the program calls into reusable libraries to take care of generic tasks

- With inversion of control, it is the framework that calls into the custom, or task-specific, code.

Dependency Injection Framework (e.g., Guice)

- The dependency injection pattern leads to code that's modular and testable, and Guice makes it easy to write
- Tell Guice how to map our interfaces to their implementations
- This configuration is done in a Guice module, a building block of an Injector

```
public class BillingModule extends AbstractModule {
  @Override
  protected void configure() {
    bind(TransactionLog.class).to(DatabaseTransactionLog.class);
    bind(CreditCardProcessor.class).to(PaypalCreditCardProcessor.class);
    bind(BillingService.class).to(RealBillingService.class);
  }
}
```

- We add @Inject to RealBillingService's constructor, which directs Guice to use it.
  Guice will inspect the annotated constructor, and lookup values for each parameter.

```
public class RealBillingService implements BillingService {
  private final CreditCardProcessor processor;
  private final TransactionLog transactionLog;

  @Inject
  public RealBillingService(CreditCardProcessor processor,
      TransactionLog transactionLog) {
    this.processor = processor;
    this.transactionLog = transactionLog;
  }
  …
```

- Finally, we can put it all together. The Injector can be used to get an instance of any of the bound classes.

```
public static void main(String[] args) {
  Injector injector = Guice.createInjector(new BillingModule());
  BillingService billingService = injector.getInstance(BillingService.class);
  ...
}
```