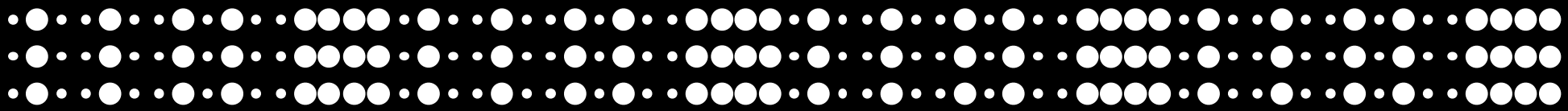


Java on Guice

Bob Lee & Jesse Wilson

What can Guice do for me?

- Write less boilerplate code
- Easy modularity
- Abstract scope
- Easy unit testing

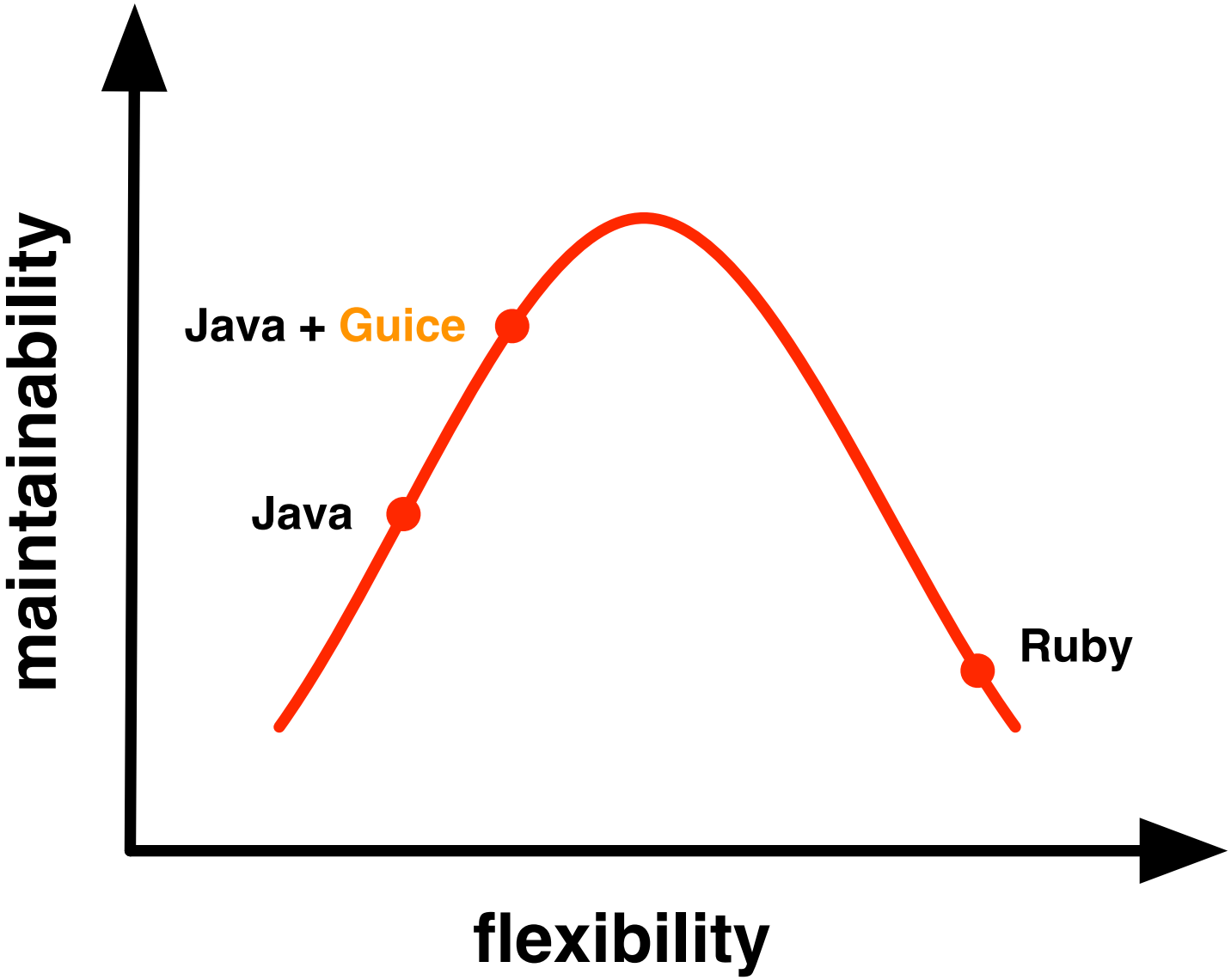
Guice's Philosophy

- Back to basics
- `@Inject` is the new "new"
  - Brevity of "new", flexibility of a factory
- Fail early but not too early
- Make it easy to do the right thing
- Types are the natural currency of Java
- Prefer annotations to convention
- Singletons aren't bad--their typical implementation is
- Focus on readability over writability
- Maximize power-to-weight ratio of API
- Balance: just because you can, doesn't mean you should

Type Safety

- Type safety != compiler errors
- Find usages
- Documentation
- Intelligent auto-completion
- Refactoring

Productivity Continuum



Guice 2

- Led by Jesse Wilson
- Coming summer 2008

An Example

- We'll show
  - Less code
  - More flexibility
  - Simpler unit test

3 actors

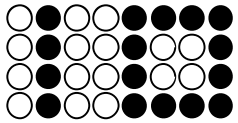
- Biller
- ShoppingCart
- A unit test

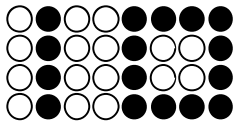
3 approaches

#1: Factory (or service locator)

#2: Dependency injection by hand

#3: Dependency injection with Guice

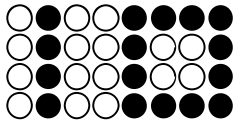


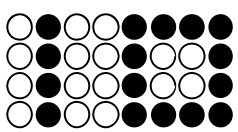


Common Code

```
public interface Biller {

  void bill(Money amount);
}


public class GoogleCheckoutBiller
    implements Biller {

  public void bill(Money amount) {
    ...
  }
}
```

```
public class MockBiller implements Biller {

  private Money amount;

  public void bill(Money amount) {
    this.amount = amount;
  }

  public Money amountBilled() {
    return this.amount;
  }
}
```

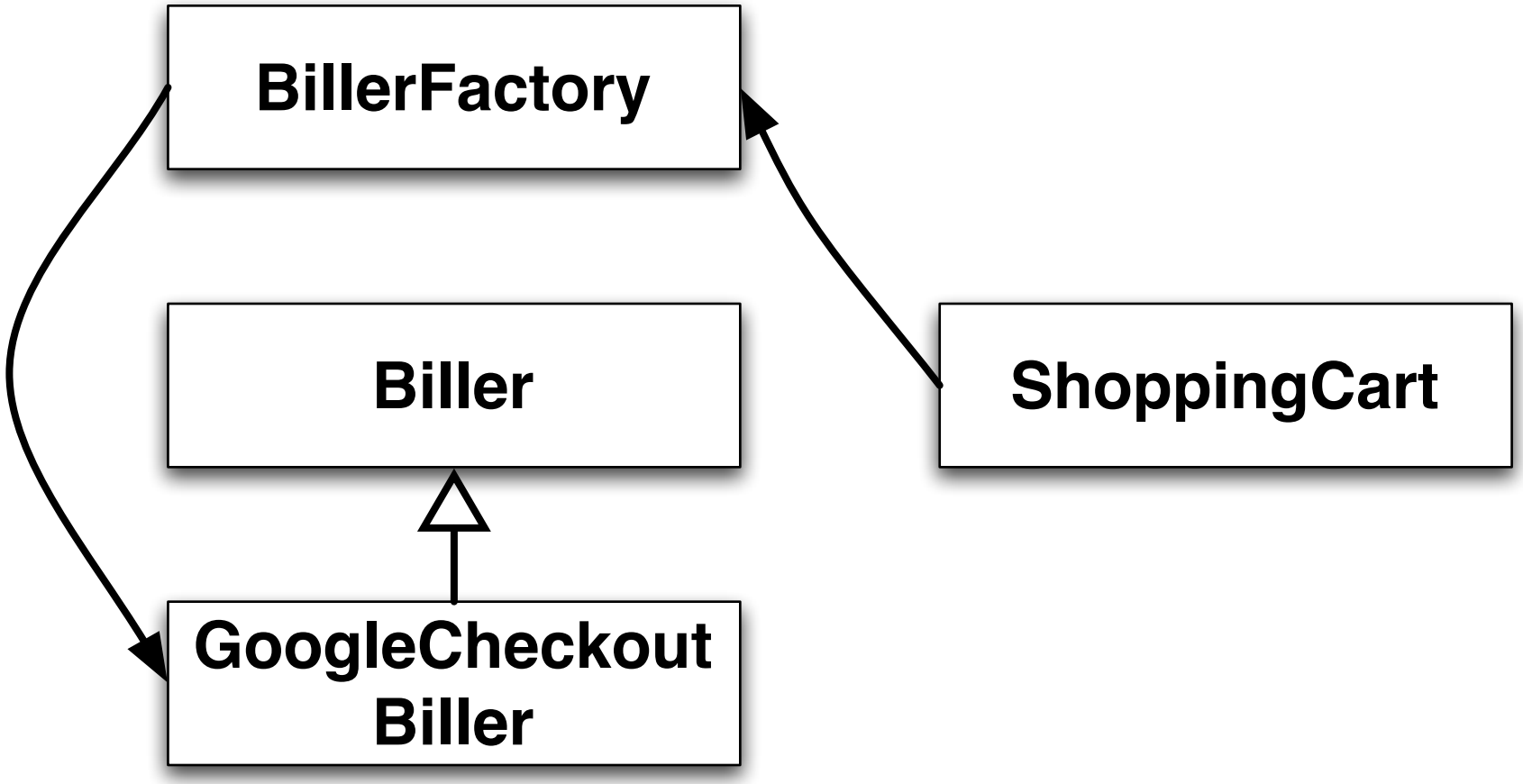




Approach #1:
Factory

```
public class BillerFactory {

  private BillerFactory() {}

  private static Biller instance = new GoogleCheckoutBiller();

  public static Biller getInstance() {
    return instance;
  }

  public static void setInstance(Biller biller) {
    instance = biller;
  }
}
```

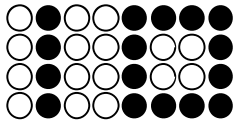
```
public class ShoppingCart {

  ...

  public void checkOut() {
    Biller biller = BillerFactory.getInstance();
    Money total = calculateTotal();
    biller.bill(total);
  }
}
```

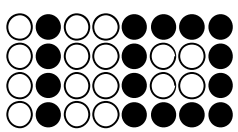

BillerFactory
Biller
ShoppingCart
GoogleCheckout
Biller

```
public void testShoppingCart() {
  Biller previous = BillerFactory.getInstance();
  try {
    MockBiller mock = new MockBiller();
    BillerFactory.setInstance(mock);
    ShoppingCart cart = new ShoppingCart();
    // Add some stuff to the cart.
    ...
    cart.checkOut();
    assertEquals(expectedTotal, mock.amountBilled());
  }
  finally {
    BillerFactory.setInstance(previous);
  }
}
```

Factory observations

- The unit test must pass the mock service to the factory and clean up afterwards
- Compile time dependency on `GoogleCheckoutBiller`
- Reusing Client in another context will be difficult
- ~60 lines of code





Approach #2:
Dependency Injection

```
public class ShoppingCart {

  private final Biller biller;

  public ShoppingCart(Biller biller) {
    this.biller = biller;
  }

  ...

  public void checkOut() {
    Money total = calculateTotal();
    biller.bill(total);
  }
}
```

```
public void testShoppingCart() {
  MockBiller mock = new MockBiller();
  ShoppingCart cart = new ShoppingCart(mock);
  // Add some stuff to the cart.
  ...
  cart.checkOut();
  assertEquals(expectedTotal, mock.amountBilled());
}
```

Before

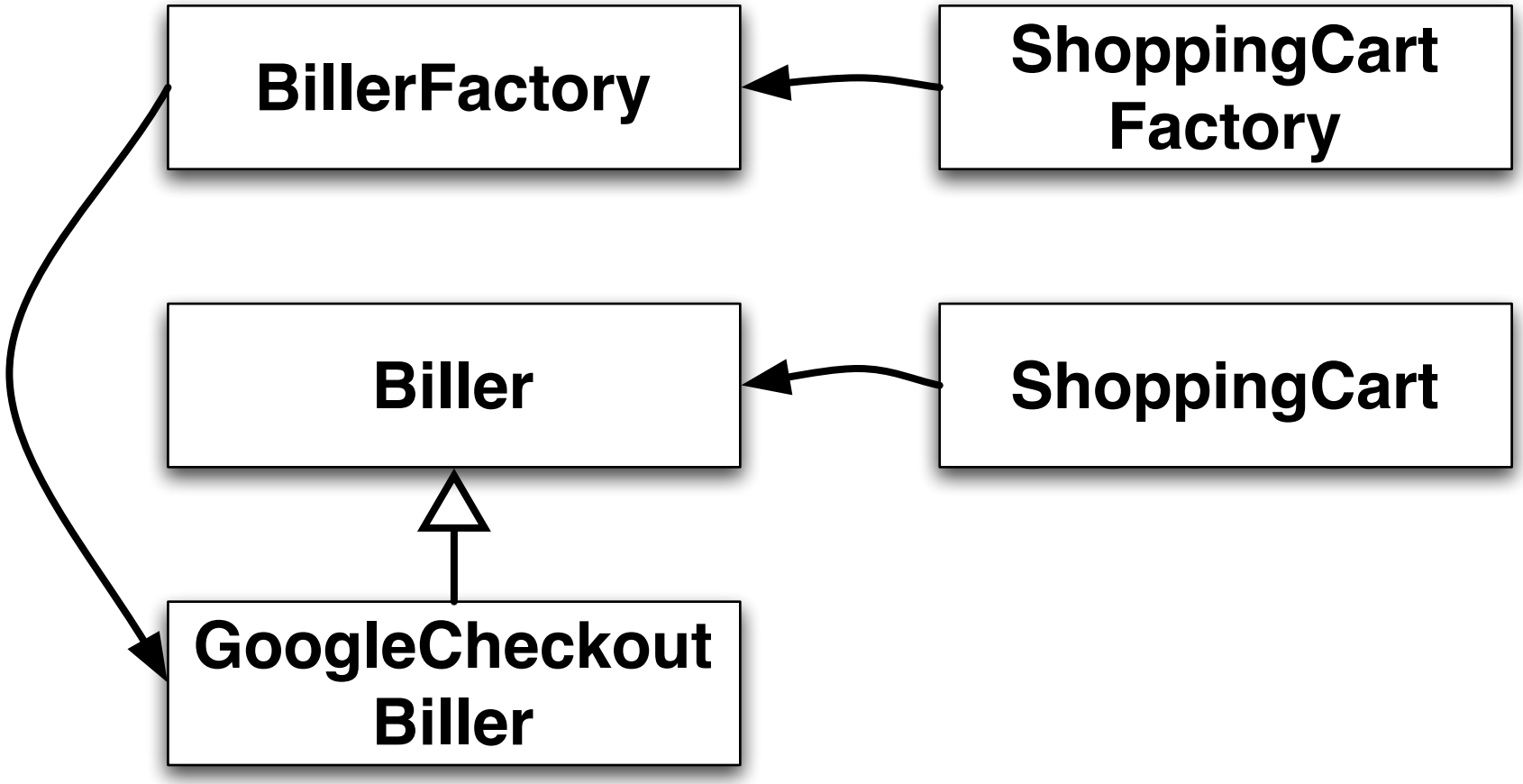
```
public void testShoppingCart() {
  Biller previous = BillerFactory.getInstance();
  try {
    MockBiller mock = new MockBiller();
    BillerFactory.setInstance(mock);
    ShoppingCart cart = new ShoppingCart();
    // Add some stuff to the cart.
    ...
    cart.checkOut();
    assertEquals(expectedTotal, mock.amountBilled());
  }
  finally {
    BillerFactory.setInstance(previous);
  }
}
```

After

```
public void testShoppingCart() {
  MockBiller mock = new MockBiller();
  ShoppingCart cart = new ShoppingCart(mock);
  // Add some stuff to the cart.
  ...
  cart.checkOut();
  assertEquals(expectedTotal, mock.amountBilled());
}
```

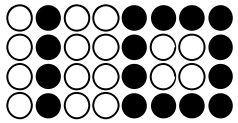
Where does the dependency go?

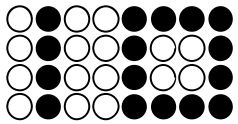
```
public static class ShoppingCartFactory {

  private ShoppingCartFactory() {}

  public static ShoppingCartFactory newInstance() {
    Biller biller = BillerFactory.getInstance();
    return new ShoppingCart(biller);
  }
}
```


BillerFactory
ShoppingCart Factory
Biller
ShoppingCart
GoogleCheckout Biller

Benefits of dependency injection

- Simpler, more reliable unit test
- Dependencies are expressed in the API
- Separates creation from usage
- Reuse classes in different contexts
- Roughly the same number of lines of code





Approach #3:
Dependency Injection with Guice

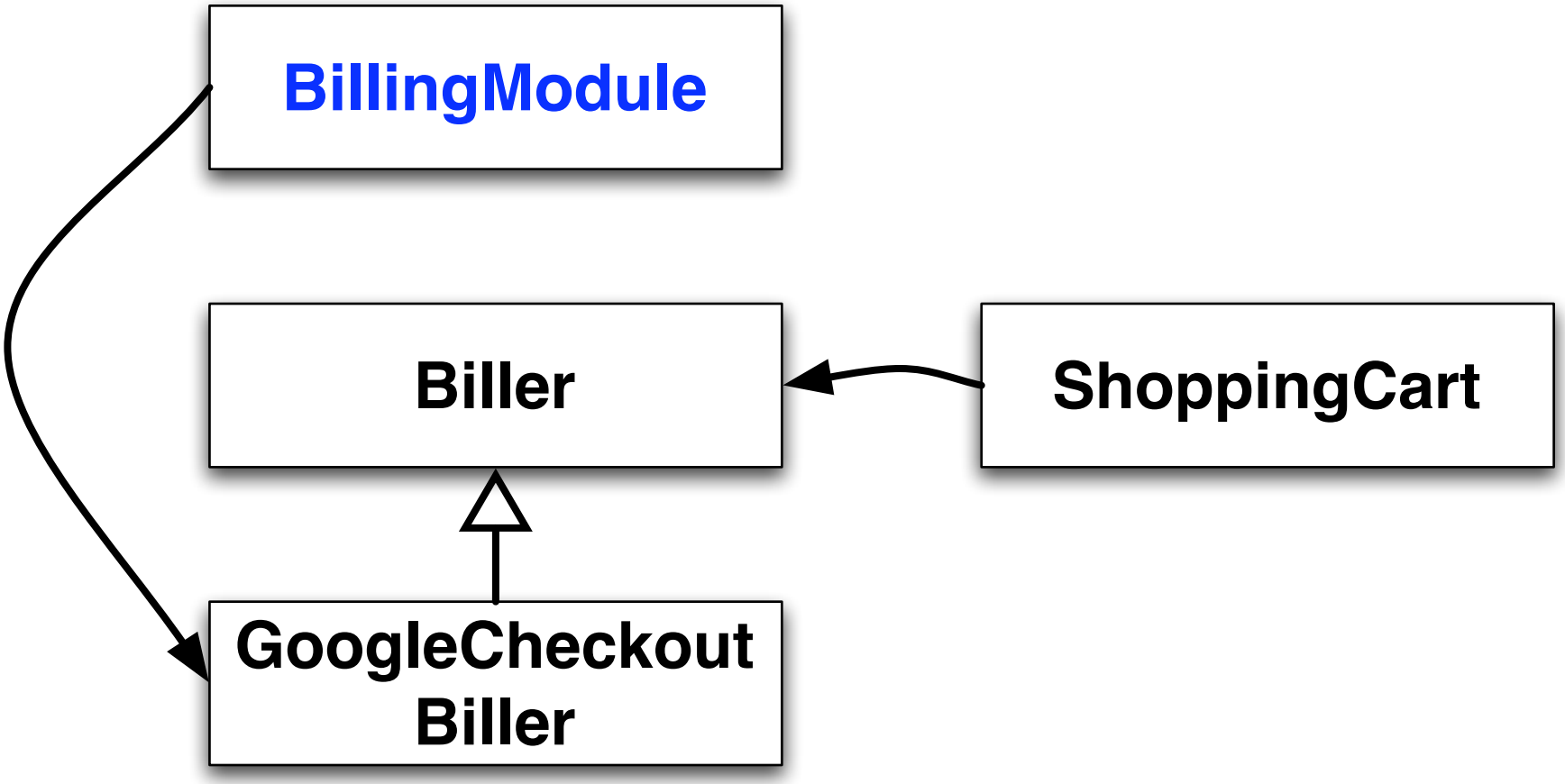
Why use a framework?

- Writing factories is tedious
  - Boilerplate code
  - Circular dependencies
  - Scopes
- More up front checking
- Make it easier to do the right thing

```
public class BillingModule extends AbstractModule {
  public void configure() {
    bind(Biller.class)
      .to(GoogleCheckoutBiller.class);
  }
}
```

```
public class ShoppingCart {

  private final Biller biller;

  @Inject
  public ShoppingCart(Biller biller) {
    this.biller = biller;
  }

  ...

  public void checkOut() {
    Money total = calculateTotal();
    biller.bill(total);
  }
}
```

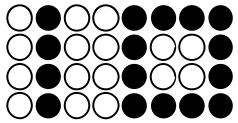
```
@Singleton
public class GoogleCheckoutBiller
    implements Biller {
  public void bill(Money amount) {
    ...
  }
}
```

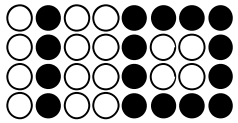

BillingModule
Biller
ShoppingCart
GoogleCheckout
Biller

```
public void testShoppingCart() {
  MockBiller mock = new MockBiller();
  ShoppingCart cart = new ShoppingCart(mock);
  // Add some stuff to the cart.
  ...
  cart.checkOut();
  assertEquals(expectedTotal, mock.amountBilled());
}
```

Conclusions

- Less framework code
  - ~20% for this example
- More startup checks
- Declarative scopes
- Better reuse





Using Guice

Bootstrapping

```
Injector injector = Guice.createInjector(
    new BillingModule(), new PetStoreModule());
PetStore store = injector.getInstance(PetStore.class);
store.start();
```

Prefer constructor injection

- Fields can be final
  - Immutability
  - Easier concurrency
- Guaranteed to be called
- See all dependencies at a glance
- There are exceptions to any rule...

Binding annotations

- Bind multiple implementations to the same type.

```
bind(Service.class)
    .annotatedWith(Blue.class)
    .to(BlueService.class);
```

```
@Inject
void injectService(@Blue Service service) {
  ...
}
```

Getting more than one instance

```
@Inject
void injectAtm(Provider<Money> atm) {
  Money one = atm.get();
  Money two = atm.get();
  ...
}
```

Binding constants

```
bindConstant()
   .annotatedWith(TheAnswer.class)
   .to("42");
```

```
@Inject @TheAnswer int answer;
```

Wire objects by hand

```
@Provides @Singleton
public Widget provideWidget(@Blue Service service) {
  return new Widget(Color.BLUE, service);
}
```

Scopes

- **Default:** No Scope
- Singleton
- HTTP Request/Session
- ...

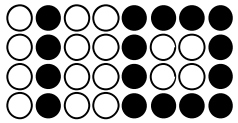
Stages

- Use DEVELOPMENT during development
- PRODUCTION during integration tests and in production
- Coming soon: TOOL

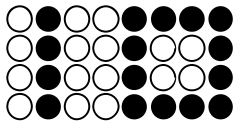
Error Handling

```
1) Error at demo.ErrorHandlingTest$BadScope.class(ErrorHandlingTest.java:1):
 Please annotate with @ScopeAnnotation.

2) Error at demo.ErrorHandlingTest$BadScope.class(ErrorHandlingTest.java:1):
 Please annotate with @Retention(RUNTIME). Bound at demo.ErrorHandlingTest
$MyModule.configure(ErrorHandlingTest.java:139).

3) Error at demo.ErrorHandlingTest$Bar.<init>(ErrorHandlingTest.java:96):
 Could not find a suitable constructor in demo.ErrorHandlingTest$Bar. Classes
must have either one (and only one) constructor annotated with @Inject or a zero-
argument constructor.

4) Error at demo.ErrorHandlingTest$Bar.bar(ErrorHandlingTest.java:100):
 Binding to java.lang.String annotated with @com.google.inject.name.Named
(value=foo) not found. Annotations on other bindings to that type include:
[@com.google.inject.name.Named(value=foo)]

5) Error at demo.ErrorHandlingTest$Bar.setNumbers(ErrorHandlingTest.java:98):
 Binding to java.util.List<java.lang.Integer> annotated with
@com.google.inject.name.Named(value=numbers) not found. No bindings to that type
were found.

6) Error at demo.ErrorHandlingTest$I.class(ErrorHandlingTest.java:1):
 java.lang.String doesn't extend demo.ErrorHandlingTest$I.
```

- http://code.google.com/p/google-guice
  - downloads, source, tutorial, users group, blogs
- Books:
  - Dependency Injection by Dhanji R. Prasanna
  - Google Guice by Robbie Vanbrabant



- http://code.google.com/p/google-guice
  - downloads, source, tutorial, users group, blogs
- Books:
  - Dependency Injection by Dhanji R. Prasanna
  - Google Guice by Robbie Vanbrabant



Questions?