```
Public class Cake {
        private double sugar;
        private double butter;
        private int eggs;
        private double flour;

        public static class Builder {
                private double sugar;
                private double butter;
                private int eggs;
                private double flour;

                public static double withSugar(double sugar) {
                        this.sugar = sugar;
                        return this;
                }
                public static double withButter(double butter) {
                        this.butter = butter;
                        return this;
                }
                public static int withEggs(int eggs) {
                        this.eggs = eggs;
                        return this;
                }
                public static double flour(double flour) {
                        this.flour = flour;
                        return this;
                }

                public Cake build() {
                        return new Cake(this);
                }
        }

        public Cake(Builder builder) {
                this.sugar = builder.sugar;
                this.butter = builder.butter;
                this.eggs = builder.eggs;
                this.flour = builder.flour;
        }

        @Override
        public String toString() {
                return "Cake [sugar=" + sugar + ", …….";
        }
}
}
```

```
Public class Application {
	public static void main(String[] args) {
		Cake cake = new Cake.Builder().withSugar(1.5).withButter(1).build();
		System.out.println(cake);
	}
}

public class ConnectionPool {
	private Connection connection = new Connection();

	public Connection getConnection() {
		return connection;
	}
}

public class ConnectionPool{
	private static ConnectionPool pool;
	private Connection connection = new Connection();

	private ConnectionPool() {
		//prevent from reflection api;
		if(pool != null) {
			throw new RuntimeException("Use getInstance() method to get the
single instance");
		}
	}

	public static ConnectionPool getPool() {
		if(pool == null) {//check for the first time
			synchronized(ConnectionPool.class) {//check for the second time
				if(pool == null) pool = new ConnectionPool();
			}
		}
		return pool;
	}

	public Connection getConnection() {
		return getPool();
	}

	protected Object readResolve() {
		return getPool();
	}
}
```

```
Public class ConnectionPool {
        private static ConnectionPool pool;
        private Connection connection = new Connection();
        private ConnectionPool () {
                if (pool != null)
                        throws RuntimeException(“Use getInstance() method to get single
instance”);
        }
        public static ConnectionPool getPool() {
                if (pool = null) {
                        synchronized(ConnectionPool.class) {
                                if(pool == null) pool = new ConnectionPool();
                        }
                }
                return pool;
        }

        public Connection getConnection() {
                return getPool();
        }

        protected Object readResolve() {
                return getPool();
        }
}
```

The factory method pattern is used to create one object and the abstract factory pattern create a family of related objects

```
Public class FWApplication {
        public static void run(Class applicationClass) {
                FWContext fwContext = new FWContext();
                try {
                        //create instance of the application
                        Object applicationObject = (Object) applicationClass.newInstance();
                        //find annotated fields
                        for (Field field : applicationObject.getClass().getDeclaredFields()) {
                                if (field.isAnnotationPresent(Inject.class)) {
                                        //get type of the field
                                        Class<?> theFieldType = field.getType();
                                        //get the object instance of this type
                                        Object instance = fwContext.getBeanOftype(theFieldType);
                                        //do the injection
                                        field.setAccessible(true);
                                        field.set(applicationObject, instance);
```

```
                    }

                }
                //call the run() method
                if (applicationObject instanceof Runnable)
                    ((Runnable)applicationObject).run();
            }
        }
    }
```

Because Spring implement AOP using proxy. This means the client that would normally talk to the target object will now talk to the proxy object in front of target object. So the code of the client we need to use the interface of the target class. Otherwise we get an exception.

Aspect Oriented Programming:
With Spring AOP we separate the logic at design time and we weave it together at runtime using a proxy. In the relative world everything seems to be separated while in reality everything is connected at its source, the unified field of pure consciousness.

Exam Feb, 2105
1. Dependency Injection is flexible way to wire objects together
2. Advantage: Flexibility. Easy to change the wiring of objects
3.

```
public class AccountService {
    private IAccountDAO accountDAO;
    public void setAccountDAO (IAccountDAO accountDAO) {
        this.accountDAO = accountDAO;
    }
    public void deposit(long accountNumber, double amount) {
        Account account = accountDAO.loadAccount(accountNumber);
        account.deposit(amount);
        accountDAO.saveAccount(account);
    }
}

<bean id="accountService" class="AccountService">
    <property name="accountDAO" ref="accountDAO"/>
</bean>
<bean id="accountDAO" class="AccountDAO"/>
<bean id="mockAccountDAO" class="MockAccountDAO"/>
```

AOP: write crosscutting concern at one place and use them at different places in your application at runtime
Advantage: SoC: separate functionality at design time but weave them together at runtime
DRY: don’t repeat yourself

```
Public class AccountService implements IAccountService {
        Collection<Account> accountList = new ArrayList();
        public void addAccount(String accountNumber, Customer customer) {
                Account account = new Account (accountNumber, customer);
                accountList.add(account);
        }
}

@Aspect
Public class TraceAdvise {
        @Before(“execution (( account package.AccountService.*(…))”)
        public void tracebeforemethod(JoinPoint join point) {
                System.out.println(“before execution of method:” +
joinpoint.getSignature().getName());
        }
        @After(“execution (( account package.AccountService.*(…))”)
public void traceaftermethod(JoinPoint join point) {
                System.out.println(“af eterxecution of method:” +
joinpoint.getSignature().getName());
        }
}
```