Spring + Hibernate

Tony Baines & Marcus Horsley

February 2011

OO Basics: Composition/Aggregation

- "has-a" relationship
- A thing that's made up of other things?
- A thing that uses other things to do work?

- Lifecycle – shared and separate
  - Composed: when I go I'm taking you with me

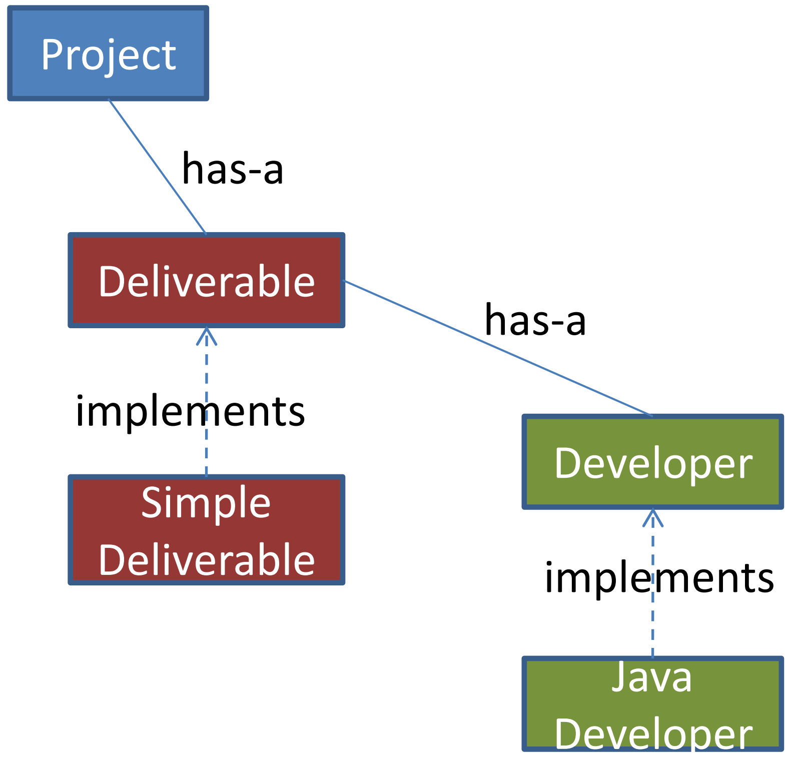
    ```
    this.thing = new Thing();
    ```

  - Aggregated: you go on without me

    ```
    public void setThing(Thing thing) {
      this.thing = thing;
    }
    ```

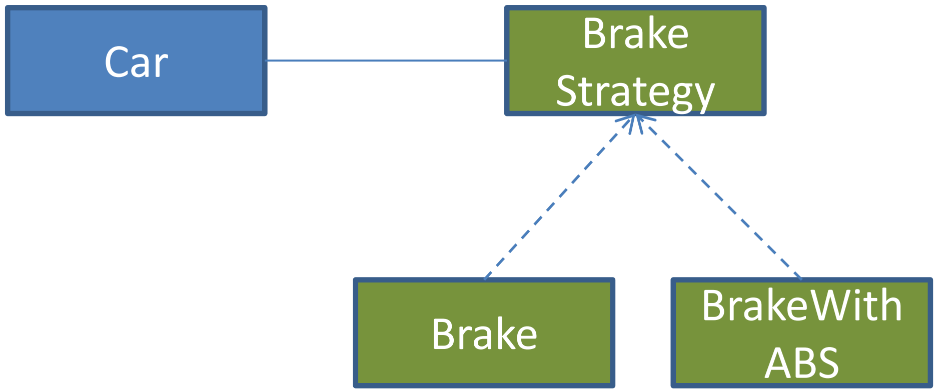
Exercise 1 (5 minutes)

- Open the Main class and build a project



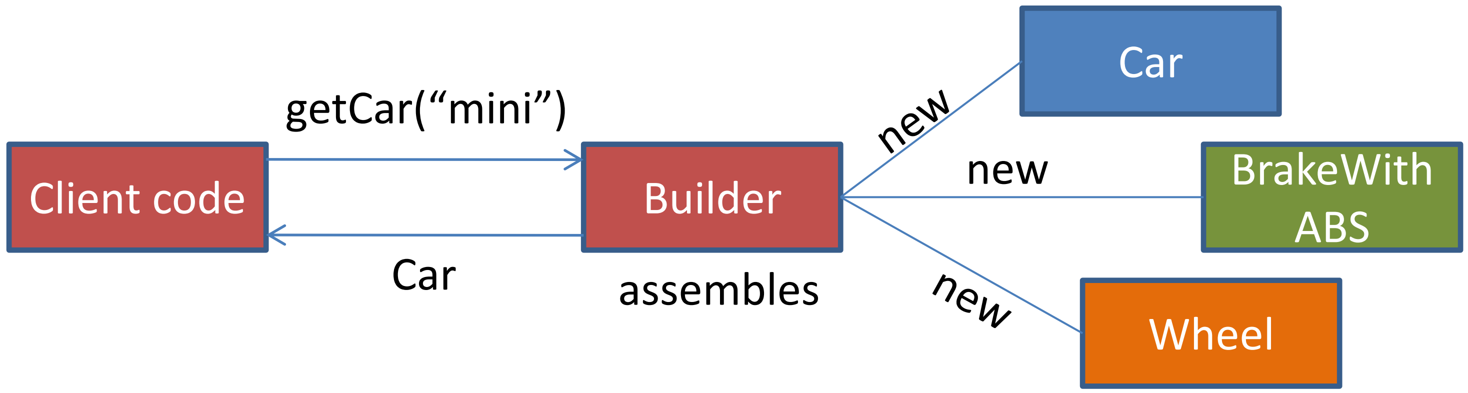
Testability & Scaling up

- Easily testable code tends to use aggregation and the injection of collaborators
  - Test doubles
- Improved design through composed behaviour
  - Strategy pattern



Building an application

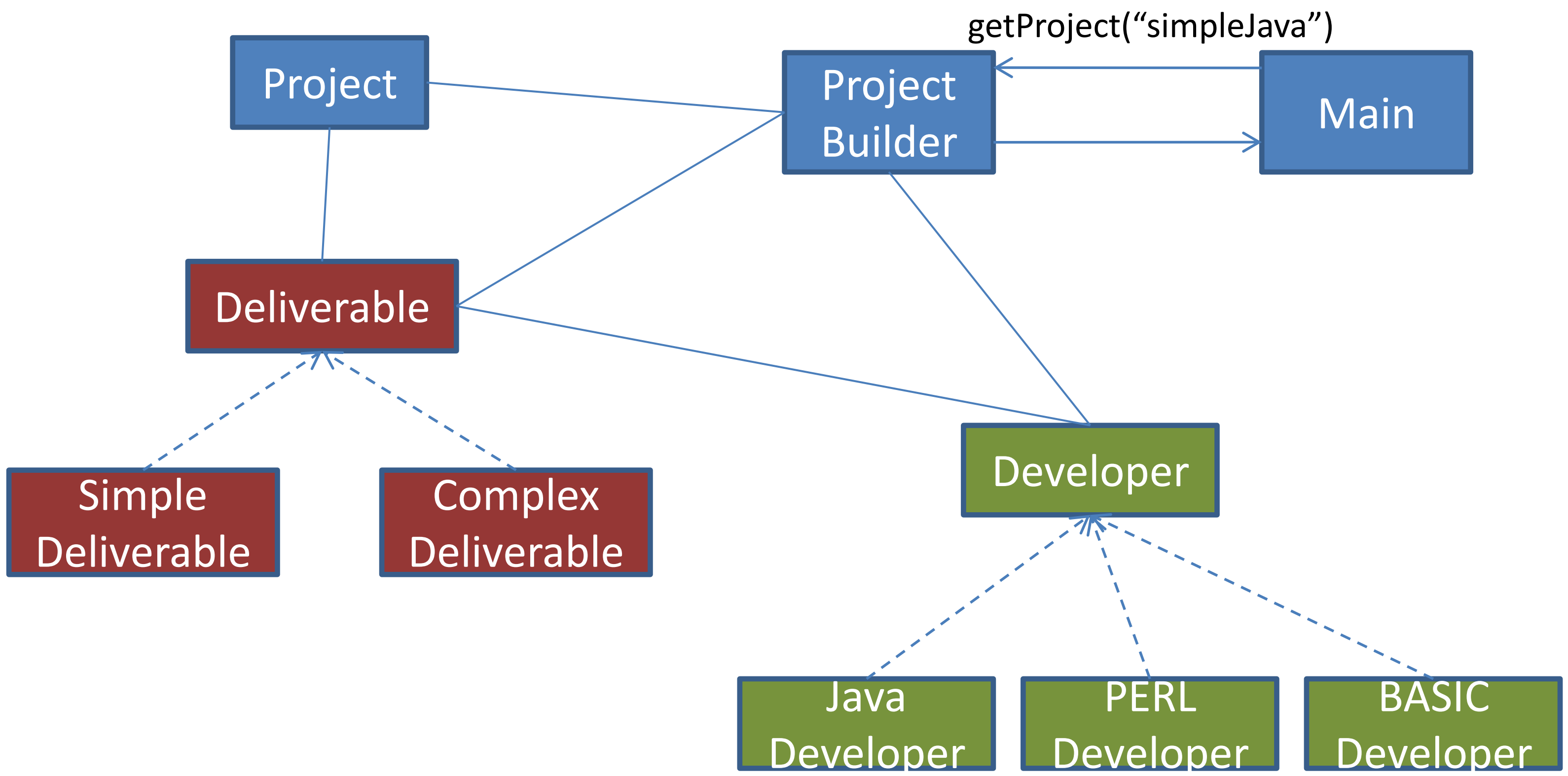
- If we don't call new in the application code, what builds the application?



- Lifecycle also needs to be managed
  - Can't reuse the same instance of Wheel
  - May want to reuse expensive objects (e.g. database connection)

Exercise 2 (10 minutes)

- Open the Main class and build a Builder

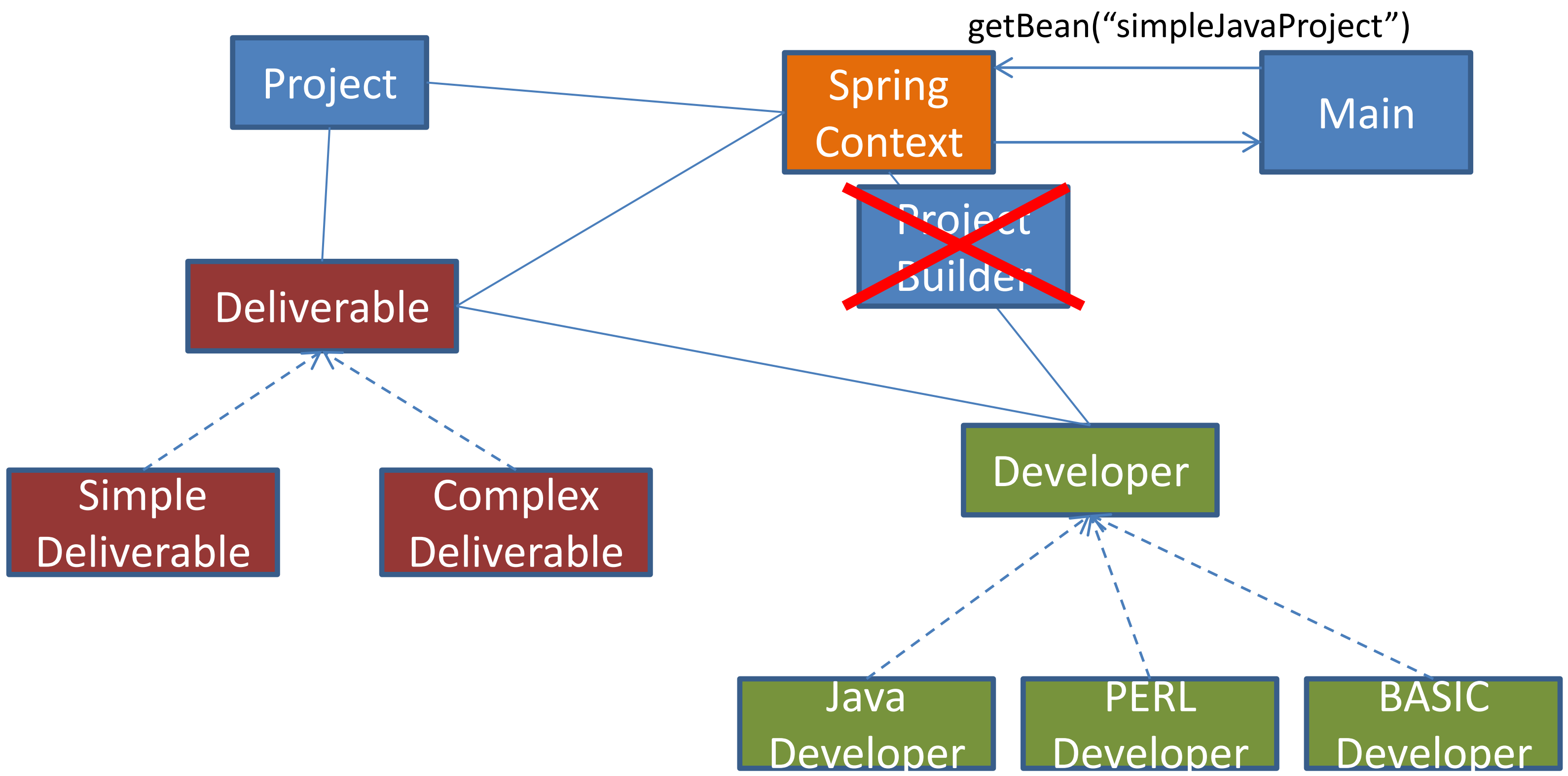


Spring has come at last

- The core of Spring is Dependency Injection (DI)
  - Lifecycle management (default is single-instance)
  - Resolve chains of dependency

Exercise 3 (10 minutes)

- Complete and extend applicationContext.xml



So what …

- We just seem to have exchanged type-safe Java code for an XML file
- Spring provides services for managed beans
  - Aspect Oriented Programming (managing cross-cutting concerns)
    - Logging, Security, Transactions
  - Framework abstraction
    - Hibernate, iBatis, JPA
  - Extensions
    - Spring MVC, Webflow, Modules, Integration

Exercise 4 (10 minutes)

- Weaving logging into our application
  - The definitions in aopDefinitions.xml are imported into the main definition
  - Examine the config and code, see what it does
    - How does it compare with hand-coding the log messages?
    - Experiment with the pointcut definitions

Reducing the clutter

- XML definitions can quickly grow
- Some of the Spring bean definitions can be inferred
  - A Car constructor declares a dependency on a Chassis, Body, Interior etc.
  - Autowiring removes some of the clutter
    - Some loss of flexibility
    - Assumes one definition of each class

Exercise 5 (10 minutes)

- Open applicationContext.xml and complete the autowired definitions.

There’s more to life than XML

- Other DI frameworks are driven by annotations
  - Google Guice, Pico-container
- Moves the definitions into Java code

Exercise 6 (10 minutes)

- Take a look at the applicationContext.xml to see the minimal config
- Open up Car.java to see examples of the annotations
  - @Component
  - @Autowired
- Is this an improvement?

Going beyond XML

- Spring 3 brought in the JavaConfig extension
- Java used to declare beans

```
<bean name="wheel"
  class="masterclass.spring.annotations.Wheel"
  scope="prototype"/>
```

Becomes

```
@Scope(“prototype”)
@Bean public Wheel wheel() { new Wheel(); }
```

- Strongly-typed – refactoring and compile-time checks without special IDE support

Exercise 7 (10 minutes)

- Wiring with JavaConfig
  - Compare the creation of the **`ApplicationContext`** with previous exercises
  - Edit **`MyApplicationContext.java`** and wire in the other dependencies

Good habits for Spring …

- Treat the bean definitions as code
  - Name things well
  - Test, test, test, …
- Use the Spring test support
  - Beans should be unit-testable in isolation
  - Integration testing
    - @ContextConfiguration – create the Spring context
    - @Rollback – cleanup changes to a database
    - @Autowired – inject the object under test
    - @Repeat / @Timed – for a test method

Good habits for Spring …

- Use the tools
  - A Spring-aware IDE will protect you from most of the simple typographic mistakes you might make
- Modularise and compose configuration
  - The XML files can grow at an alarming rate
  - Use `<import …/>` to modularise e.g. by layer
- Not everything needs to be a Spring bean
  - Pure Java Builders, Factories etc. can manage objects that don’t need the Spring special treatment

Good habits for Spring …

- Use Autowiring carefully
  - It reduces XML, but it makes configurations less explicit
- Prefer constructor injection over setters
  - Although setter-injection is clearer you’ll need to take additional steps to make sure the class still works (or has meaningful errors) if not all properties are set
- Prefer constructor `type=` to `index=`
  - Less brittle to change
  - Use index to remove ambiguity where necessary

Good habits for Spring …

- Use the shortcut forms

```
<property name="meaning" value="42" />
<constructor-arg ref="deepThoughtDAO" />
```

- Use inner beans where appropriate
- Use abstract beans to reduce duplication
  - Inherit common configuration `parent="…"`
- Prefer `id=` over `name=` to trap accidental name conflicts