HOW IT IS DONE IN JAVA

Design Patterns and Software Engineering Techniques for Java, and how they may apply to Chisel

CS 294-88: Declarative Hardware Design

Martin Maas, 03/05/2013

Introduction

- Java has been very popular language in production use for a long time (especially enterprise applications).
- Significant performance overheads (GC, JIT,...), yet people are using it for big, mission-critical applications.
- Reason (arguably): Enforces rigorous design methodology; availability of a large set of design patterns and tools.
- Many of these should be applicable to Chisel (Scala is fundamentally similar to Java, concepts may carry over).
- Much of this will be familiar to you already - this talk is intended to organize the content and create a basis for discussion of how to use these techniques in Chisel.

Talk Organization

1. Design Strategies
2. Dependency Injection
3. Testing & Mocking
4. Java Tools

Part I: Design Strategies

Strategy 1: Use Interfaces whenever feasibly possible

- Naïve Strategy: When using a component, refer to it by its class type and create an instance of that class.

```
Foo foo = new Foo();
```

- Problem #1: What if we suddenly need to use a different implementation (e.g. for testing)?
  - Aside: Could use inheritance, but this breaks abstraction - ideally, declare all classes that are not specifically designed to be inherited from as `final`.
- Problem #2: What if we have a class that wants to provide multiple functionalities?
- For example: Consider some class `SignalToImageDSP` representing a pipeline stage that takes signal data and produces image data.

Strategy 1: Use Interfaces whenever feasibly possible

- Use when initializing a pipeline:

```
SignalDSP prevStage = new SignalDSP();
SignalToImageDSP stage =
    new SignalToImageDSP(prevStage);
ImageDSP nextStage = new ImageProc(stage);
```

- Say we decide at some point that we want to use a truncated version of the pipeline, where the image data is approximated without using sensors. But `ImageProc`'s constructor expects a `SignalToImageDSP`...
- Inheritance? Bad idea: (i) have to allow inheriting from `SignalToImageDSP`, breaks abstractions and makes code unmanageable (we change something in `SignalToImageDSP`, have to change all its children), (ii) spurious methods with undefined behavior (e.g. the child would still accept a signal).

Strategy 1: Use Interfaces whenever feasibly possible

- Better: Use interfaces. Can now easily use the classes interchangeably, based on specific functionality/type.

```
class SignalToImageDSP implements
    SignalConsumer, ImageProducer {...}
class SignalToImageTest implements
    ImageProducer {...}
```

- Can add functionality to a different class; e.g. to log all signals, can make a class `Logger` a `SignalConsumer` and use it in the existing pipeline without changes to any other classes (this is called a *mixin*).
- *Important*: Do not turn classes into interfaces once you have to (then it's often too late). Design around interfaces and functionalities from the start.

How does this apply to Chisel?

- Interfaces are (arguably) the software-equivalent to bundles of wires with a specific functionality.
- Allow defining interfaces in Chisel which consist of a bundle of wires and a well-documented contract.
- When instantiating a component, make its type the one of the relevant interface in which capacity it is going to be used.
- If used in multiple capacities, can use dependency injection to return the same object with different types (see later).

```
BTB implements BranchConsumer, TargetPredictor
PHT implements BranchConsumer, BranchPredictor
BranchConsumer = IF(in: pc, taken, val; out: rdy)
BranchPredictor = IF(in: pc, val; out: taken, val)
```

Strategy 2: Static Factory Methods

- Naïve Strategy: To instantiate a component, use one of its constructors (e.g. `A a = new A(42);`).
- Problem #1: What if you have multiple ways to instantiate a component, which have the same signature (e.g. for a Mem, one takes the number of read ports/write ports, one width and depth - both have signature `(int, int)`).
- Problem #2: Readability; What is the meaning of a specific constructor - need to look it up documentation.
- Problem #3: What if we optionally want to reuse a component instead of creating a new one?
- **Solution**: Use static factory methods instead of constructors.

Strategy 2: Static Factory Methods

```
public class Mem {
  private Mem() {...} // private constructor
  public static Mem fromPorts(int r, int w) {
    Mem m = new Mem();
    m.readPorts = r;
    m.writePorts = w;
    return m;
  }
  public static Mem fromDims(int d, int w) {
    Mem m = new Mem();
    m.depth = d;
    m.width = w;
    return m;
  }
}
```

Strategy 3: Minimizing accessibility

- Hide all fields, methods, classes that are not part of an external interface. Note: `protected` variables are part of the public interface and often commit to implementation details. In many cases, package-private is a better fit.
- Think in terms of contracts - forbid everything that you do not want to explicitly allow (subclassing, instantiating class using constructor, accessing static fields). If something is possible (i.e. type checks), people will potentially use it, and it becomes part of the contract (introducing coupling).
- Don't expose implementation details to the outside world (e.g. Chisel should not expose its internal representation).

Part II: Dependency Injection

Dependency Injection Basics

- Software systems consist of a large number of components with dependencies between each other.
- These dependencies are often hidden in the code and hard to change or customize as the system grows.
- A simple example[1]:
  - A web application `MyWebApp` uses a billing service to charge users for services (`BillingService`).
  - In its first iteration, only credit cards are supported, and handled by a `CreditCardService` class.
  - Say this service has to take some configuration data, such as `accountNo`, `companyName` and `vendorId`.
- `CreditCardService` is a dependency of `BillingService`.
- `BillingService` is a dependency of `MyWebApp`.

[1] Inspired by `https://code.google.com/p/google-guice/wiki/Motivation`

First attempt: Constructor Calls

- Naïve approach: Classes instantiate their dependencies.

```
public final class BillingService {
  private int accountNo = 1234;
  private String companyName = "ACME";
  private String vendorId = 42;

  private CreditCardService myCreditCardService;
  public BillingService() {
    myCreditCardService =
        new CreditCardService(accountNo,
            companyName, vendorId);
  }
}
```

First attempt: Constructor Calls

- Problem #1: Configuration parameters are hidden away in the code - very inflexible to change.
- Problem #2: How can we replace `CreditCardService` by a different implementation (e.g. for testing purposes)?
- Problem #3: We have a compile-time dependency: `CreditCardService` must be available at compile time.
- **Solution**: Use factory (configuration) objects to encapsulate dependencies and configuration parameters.
- *Note: For readability, all interfaces are prefixed with $I^\star$.*

Second attempt: Factories

- Create a factory to encapsulate instantiation of objects, and configuration objects to store parameters (can be combined for brevity, but better keep separate).

```
public final class CCServiceFactory {
  private int accountNo = 1234;
  private String companyName = "ACME";
  private String vendorId = 42;

  public ICreditCardService makeCCService() {
    return new CreditCardService(accountNo,
        companyName, vendorId);
  }
}
```

Second attempt: Factories

```
public final class BillingService {
  private CreditCardServiceFactory factory;
  private ICreditCardService myCreditCardService;

  public BillingService(CCServiceFactory f) {
    myCreditCardService = f.makeCCService();
    factory = f;
  }
}
```

- Expose configuration parameters.
- Can implement different factories for different configurations.
- Can supply factory at run-time.

Second attempt: Factories

- Problem #1: Dependencies are still hidden in the code - not clear where which class is instantiated.
- Problem #2a: Need to pass factories down through component hierarchy (so it reaches applicable classes); introduces coupling between unrelated components:

```
public final class MyWebApp {
  private BillingService billingService;
  public MyWebApp(CCServiceFactory f) {
    billingService = new BillingService(f);
  }
}
```

Second attempt: Factories

- Problem #2b: If everything is instantiated by factories, they need to call each other (i.e. depend on each other) - managing factories across an entire codebase can be very difficult.
- Problem #3: It is not always clear which instance of a class is required - what if some place in the application needs a credit card service, but another a Pay Pal service? (Factory methods like `makeMainPaymentService`, `makeAltPaymentService`, etc. are error-prone - what do they mean in each context?).
- **Solution**: Make dependencies explicit in the signature of the constructor.

Third attempt: Manual Dependency Management

- Take dependencies as constructor arguments:

```
public final class MyWebApp {
  private IBillingService billingService;
  public MyWebApp(IBillingService b) {
    billingService = b;
  }
}

public final class BillingService {
  private ICreditCardService myCCService;
  public BillingService(ICreditCardService s) {
    myCCService = s;
  }
}
```

Third attempt: Manual Dependency Management

- When instantiating the top-level module, need to instantiate the entire component hierarchy:

```
public static int main(String[] args) {
  MyWebApp app = new MyWebApp(
      new BillingService(
          new CreditCardService(accountNo,
              companyName, vendorId)));
}
```

- It solves all the problems mentioned above, but too messy to be practical for applications with hundreds of components.

Dependency Injection

- This is where Dependency Injection comes in.
- Dependency Injection is similar to the third approach, but uses an automated framework to handle instantiation.
- Main Idea: Generate a configuration class that contains a set of rules to describe how to resolve each dependency. When the framework instantiates a class, it will recursively resolve all dependencies.
- Popular Dependency Injection Frameworks for Java: Spring, **Google Guice**, Pico Container.

Dependency Injection: Example (Google Guice)

- Step I: Annotate every constructor that has dependencies which should be automatically resolved by Guice:

```
public final class BillingService {
  private ICreditCardService myCCService;

  @Inject
  public BillingService(ICreditCardService s) {
    myCCService = s;
  }
}
```

Dependency Injection: Example (Google Guice)

- Step II: Create a Guice configuration that **binds** types to a specific implementation:

```
public class MyModule extends AbstractModule {
  @Override
  protected void configure() {
    bind(IBillingService.class).
        to(BillingService.class);

    @Provides // Perform complex initialization
    ICreditCardService provideCCService() {
      return new CreditCardService(accountNo,
          companyName, vendorId)
    }
  }
}
```

Dependency Injection: Example (Google Guice)

- Step III: Tell Guice to instantiate the top-level class - all other dependencies will be resolved automatically.

```
public static int main(String[] args) {
  Injector injector =
      Guice.createInjector(new MyModule());
  MyWebApp app =
      injector.getInstance(MyWebApp.class);
}
```

Advanced Dependency Injection

- This is only the tip of the iceberg...
- Feature #1: If you need different instances of the same type, use annotations to distinguish.

```
public final class TestBillingService
    implements IBillingService {
  private ICreditCardService myCCService;

  @Inject
  public TestBillingService(
      @CCTest ICreditCardService s) {
    myCCService = s;
  }
}
```

Advanced Dependency Injection

```
public class MyModule extends AbstractModule {
  @Override
  protected void configure() {
    bind(IBillingService.class).
        annotatedWith(CCTest.class)
        to(BillingService.class);
  }
}
```

- Can use generic `@Named("someString")` attribute, or custom attributes that can be entire class of configuration parameters.

Advanced Dependency Injection

- Feature #2: If you want to use a factory, Guice can generate it for you (from its rules).
- Feature #3: Can define scopes to determine when created objects should be reused and when to create a new one.
- Feature #4: Inject not only classes, but your configuration parameters, too.
- Feature #5: Guice can plot the resulting object graph for you.
- Feature #6: Guice supports a plug-ins to add functionality.

How does this apply to Chisel?

- Instead of using configuration objects and query them from within your code, can encapsulate all parameters in a configuration for a dependency injection framework.
- Let the dependency injection framework construct the design.
- Advantages:
  - More flexibility for design space exploration: cover both design parameters and dependencies (e.g. easy to replace one component by another without changing code anywhere - in fact, without even having planned replacing that component in the initial design).
  - Can capture all design parameters in one place and reuse them across components (structure them whichever way you want, without thinking about the design or having to pass them around when constructing the design).
  - Allows for a lot of introspection into design points (e.g. plot component graph for a particular design point).

Part III: Testing & Mocking

Testing in Java

- **JUnit** de facto standard testing framework in Java
- Your standard xUnit-style testing framework with `setUp()`, `tearDown()`, `testX(...)`,...
- Types of testing:
  - Black-box testing: test the exposed interfaces (instantiate component, provide test vectors, mock out any dependencies).
  - White-box testing: test the specific implementation (extend the component to test for correctness of internal state, etc. One possible strategy: inheritance + *package-private*). *Some people disagree with this type of testing.*
  - Integration tests: put multiple components together and test their interactions (Dependency Injection!).
- Measure test coverage (more difficult for hardware?)

Making a design testable

- Key strategy: reduce coupling. Make it possible to test a component in isolation by mocking out everything around it.
- **Mock object**: An object that implements the interface of a real component, but fakes functionality and checks input.
- Dependency injection can be used to instantiate components and mock objects for tests.
- Put tests into a separate directory hierarchy, but keep them in the same package, so they can access package-private and protected fields.
- How to generate mock objects? Writing them by hand...
  - ...separates the mock object code from the test code, making it less clear to see what we are testing/expecting.
  - ...requires rewriting and maintaining lots of boilerplate code.

Mocking Frameworks

- A mocking framework takes care of creating mock objects for you. Examples for Java: **EasyMock**, jMock.[2]
- Basic idea: Takes an interface, implements it for you and allows you to:
  - Define which function calls to that interface are expected during the test and how to respond to them.
  - Check (at the end of the test) whether the calls that were actually received match these rules.
- The rules are defined in-place in the test code - no need to create a new class or write setup/tear down code.

[2] For C++, check out googlemock (`https://code.google.com/p/googlemock/`).

EasyMock Example

- Remember the previous example with the credit card processor. We want to test the `BillingService` without performing a real credit card transaction.
- We mock out the credit card service and replace it by a mock object implementing the same interface.
- Consider the following interface for the credit card service:

```
public interface ICreditCardService {
  int getTransactionMax();
  boolean charge(int amount, String account);
}
```

EasyMock Example

- Assume we want to test the following (nonsensical) method of the billing service:

```
public final class BillingService {
  private ICreditCardService myCCService;

  public boolean charge(int amount, String acct) {
    if (amount > myCCService.getTransactionMax())
      return false;
    return myCCService.charge(amount, acct);
  }
}
```

EasyMock Example

- In the test case for the billing service, create mock object during setup...

```
public class BillingServiceTest {
  ICreditCardService mock;
  BillingService b;
  @Before public void setUp() {
    mock = createMock(ICreditCardService.class);
    BillingService b = new BillingService(mock);
  }
}
```

EasyMock Example

- ...and state the expected function calls and their return values.

```
@Test public void testSuccessfulCharge() {
  // Note: Constants should be predeclared
  expect(mock.getTransactionMax()).andReturn(1000);
  expect(mock.charge(500, "1234")).andReturn(true);
  replay(mock);
  assertTrue(b.charge(500, "1234"));
  verify(mock);
}
```

Advanced EasyMock

- Say range how many calls are expected, change over time

```
expect(mock.getTransactionMax()).
    andReturn(100).times(3).andReturn(0);
```

- Consider or ignore order of method calls
- More general argument matchers (e.g. `isNull`, regex):

```
expect(mock.charge(or(lt(100), eq(500)),
    startsWith("act"))).andReturn(true);
```

- Capture parameters, generate result using functions:

```
expect(mock.getTransactionMax()).
    andAnswer(new IAnswer<Boolean>() {
  public Boolean answer() throws Throwable {
    return false;
  }});
```

- Partial mocking (overwrite only certain methods)

How does this apply to Chisel?

- Mock objects are the software equivalent of mock components in test harnesses.
- EasyMock would translate almost directly into Chisel: could define expectations for input signals, and the resulting output signals.
- Would allow to write much more readable test code and isolate components better: allows to write individual test cases as single component.
- Using JUnit would allow Chisel to build on an existing testing infrastructure, integrate it with the build system (sbt supports testing) and would also allow design verification.
- Requirement: Allow testing within the JVM, without requiring the C++ emulator.

Part IV: Java Tools

Build System

- Chisel (partly) still uses Makefiles, but there are much more powerful build systems out there, e.g. ant, sbt, Maven,...
- Advantages over Makefiles:
  - Easier to check for correctness and process automatically (Makefiles are largely bash commands), i.e. less error-prone.
  - Less fragile and clumsy than Autoconf/Automake.
  - Use library of functions (e.g. create tar/gz file) instead of having to write custom commands.
  - Build system understands compiler options (less risk of getting compiler parameters wrong).
  - No interference with environment variables.
- Specifically sbt: Download dependencies automatically, configuration is written in Scala DSL, incremental compilation, continuous compilation/triggered execution
- Integrate automatic testing into build.

Documentation

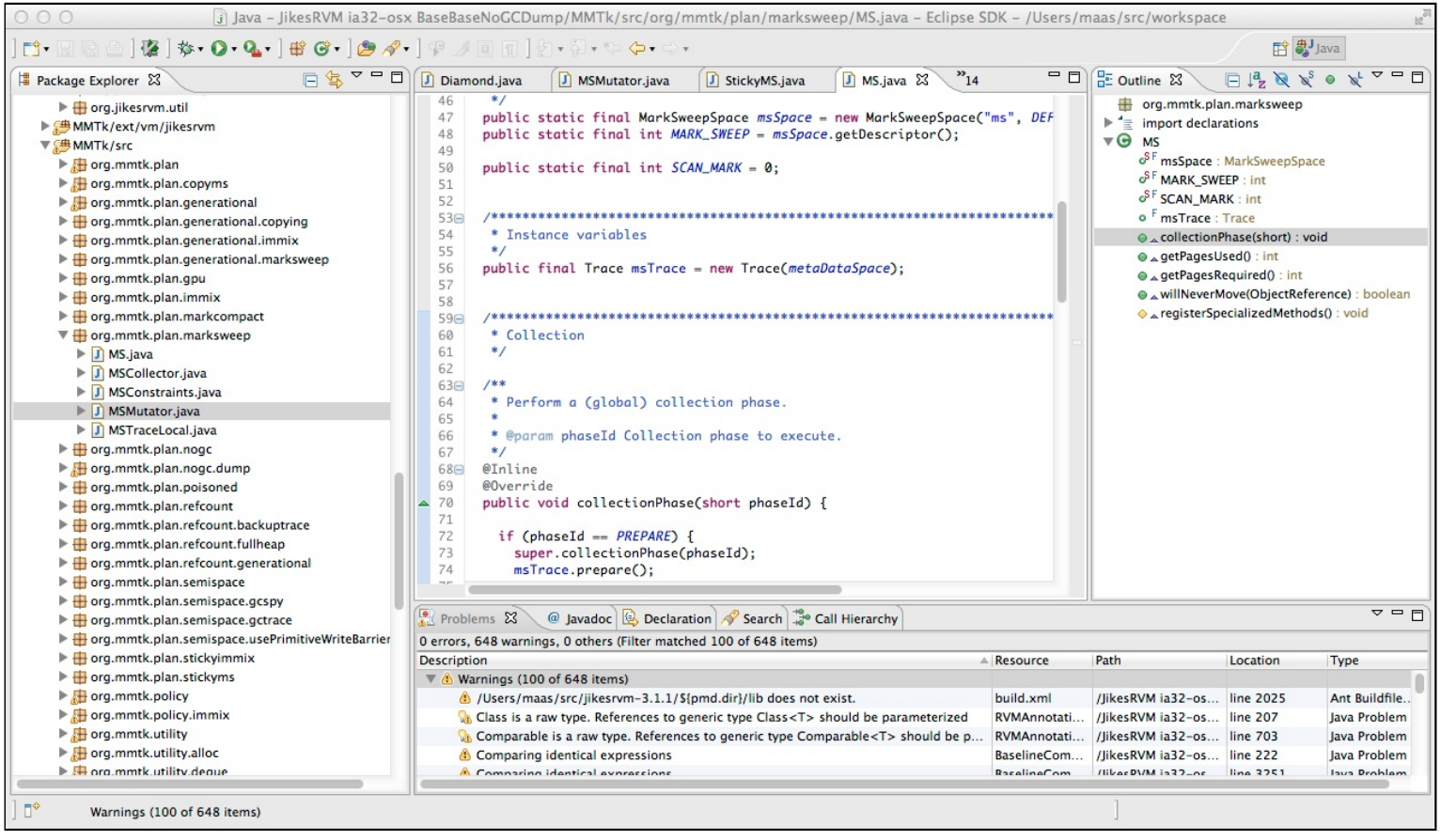
- Java makes extensive use of JavaDoc:

```
/**
* Get the project name or null
* @param event the event
* @return the project that raised this event
* @since Ant1.7.1
*/
```

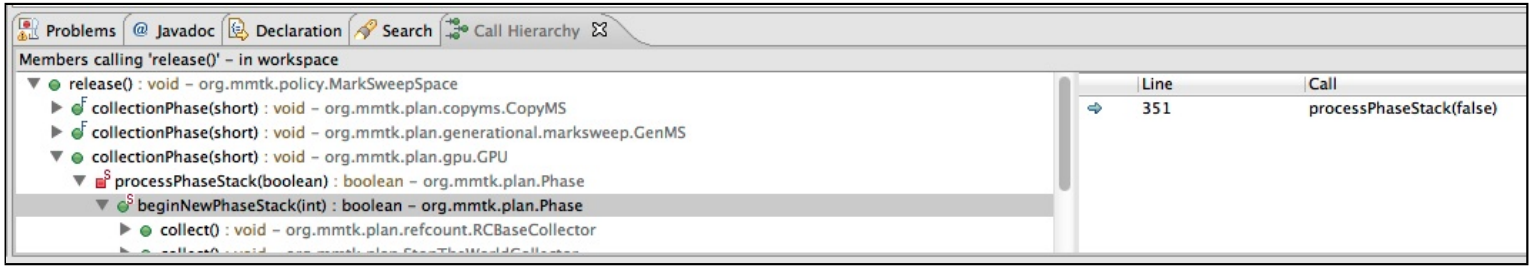
- Code is used to print documentation, but also to display tool-tips in your IDE and navigate the code.
- Even allows you to link between classes and comment (if you click on a link in a comment, you jump to the right class).
- In Chisel, this could be used to tell you the role of every wire, and the contract for using them - potentially even creating an annotated/interactive schematic.

Eclipse

- System developers often don't like GUIs. And yes, some of them are slow and cumbersome...
- But for high-languages languages like Java, they can boost productivity significantly.

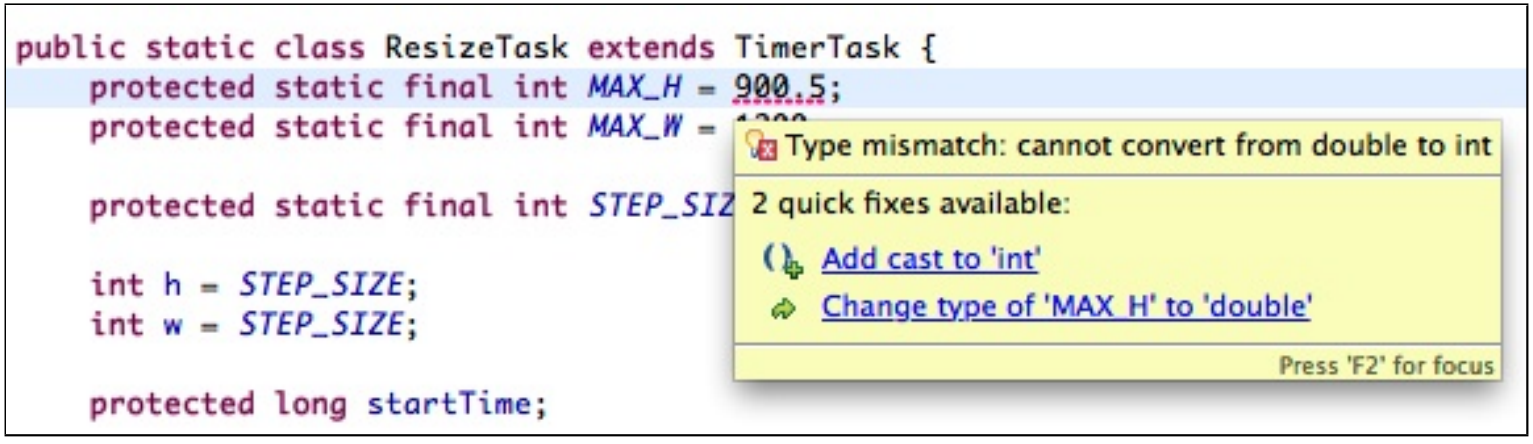


Eclipse



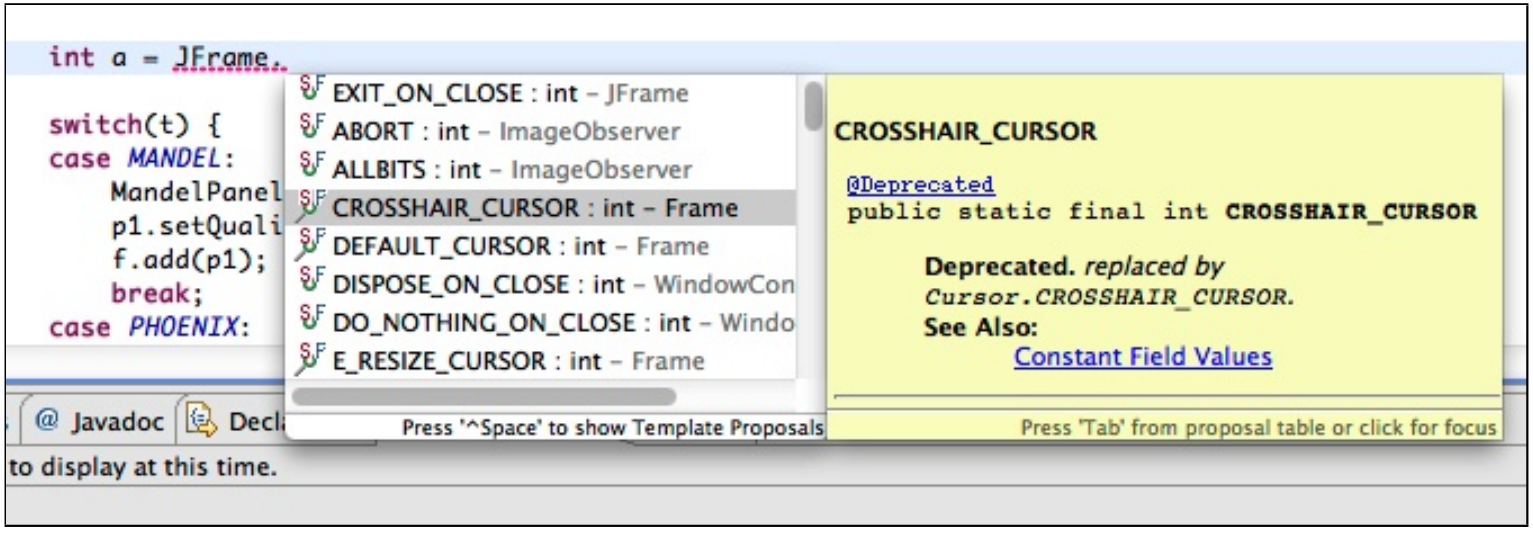
Browse code by high-level structure, instead of files (e.g. object hierarchy, call hierarchy, function overridden by the current one, all places where a function is instantiated/called) - especially powerful in combination with keyboard shortcuts.

Eclipse



Code compiles in the background - get more errors immediately.
Incremental compilation.

Eclipse



More sophisticated auto-complete (uses compile-time information).
Integrated documentation: JavaDoc parsed at run-time and shown as tool-tip.

Eclipse

- Further Advantages:
  - Build, testing, version control, debugging etc. can all be managed from one place.
  - Manages import statements for you: no forgotten includes.
  - Automatic code generation and organization (e.g. refactoring, implementing interfaces,...).
- Especially for Chisel, this could be very useful - build circuit in background, show e.g. combinational loops, invalid number of ports, etc. Or show all places where a wire is used...
- There is a vim plugin for Eclipse - and with the right color setting, it looks almost the same ;-).

Conclusion

Conclusion

- There is a large number of production-quality tools used by the Java (and Scala) community.
- Many of them make use of the fact that Java is a high-level language and leverage this to lead to clearer and more maintainable designs, better test coverage,...
- Should leverage these tools for Chisel!
- Particularly (aka my Chisel wish list):
  - Use notion of interfaces to decouple component implementation from API/use.
  - Use Dependency Injection to isolate configuration/design space exploration from behavior code.
  - Use JUnit/EasyMock to build testing infrastructure that makes component tests easy to write and keep them in one place.
  - Integrate these components into a workflow using a good build tool (sbt?), Eclipse plugins and documentation of code.

References

Examples lifted from the following books and documents (any errors are, of course, my own):

- Joshua Bloch. 2008. Effective Java (2nd Edition) (The Java Series) (2 ed.). Prentice Hall PTR, Upper Saddle River, NJ, USA.
- Google Guice Documentation

  `https://code.google.com/p/google-guice/wiki/GettingStarted`
- EasyMock Documentation

  `http://www.easymock.org/EasyMock3_1_Documentation.html`