OMIS 3.0 Middle Tier Standards - Draft

AUTHORS:

Stephen Abson

Introduction

Objectives

The objectives of OMIS 3.0 are as follows:

- Develop rich, framework independent domain model with loose coupled entities.
- Develop business logic in the middle tier.
- Develop RDBMS vendor independent persistent tier.
- Incorporate persistence framework and dependency injection.
- Place all persistence operations in persistence tier.
- Place all presentation logic in presentation tier.
- Modularize application features allowing them to be optionally included in build.
- Separate middle tier concerns using AOP – transactions, security, logging, etc.
- Develop framework independent APIs and business logic.
- Standardize security roles and permissions.

Specifications Used

- *The Java Language Specification*, Java SE 7 Edition (http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf)
- *The JavaBean Specification*, Version 1.0.1 (http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html)
- *Java Servlet 3.0* (http://download.oracle.com/otndocs/jcp/servlet-3.0-fr-eval-oth-JSpec/)
- *Java ServerPages 2.2* (http://download.oracle.com/otndocs/jcp/jsp-2.2-mrel-eval-oth-JSpec/)

Other Documents Used

- *Effective Java - Second Edition*, Joshua Bloch (2008)
- *Java Generics and Collections*, Maurice Naftalin and Philip Wadler (2006)
- *Java Persistence with Hibernate*, Christian Bauer and Gavin King (2006)
- Spring Framework and Spring Security Reference Manual (http://www.springsource.org/documentation)
- Hibernate Reference Manual (http://www.hibernate.org/docs)

Architecture

- Platform - Java 1.8

Frameworks and specifications used in Montana Implementation

- Web – Servlet 3.1/JSP 2.2
- Web View - JSTL 1.2, Spring Forms, jQuery
- MVC, AOP, Dependency Injection, Project Management - Spring 4.x
- Security – Spring Security 3.1
- Persistence Provider - Hibernate 4.x
- Reports – JasperReports

Java

General Java Best Practices

Use parameterized types

Parameterized types serve to strengthen Java's compile time type checking by allowing the actual types expected to be used in a module to be specified as a parameter to the type declaration. This eliminates the need to explicitly cast returned objects and reduces the risk of runtime `ClassCastException`s been thrown. The end result is less errors in less code.

The transition to using parameterized types is part of what is known as *generification*. In Java, generics allow additional type information to be specified at compile time and for the compiler to strictly enforce stronger type checking where specific type information is available.[1]

Recommendation

Use a parameterized type wherever possible. A parameterized type allows the actual type to be specified in angled brackets (less and greater than symbols – for example `<Integer>`):

```
// Good, generified code using parameterized type
List<Offender> offenders = new ArrayList<Offender>();
```

In the above example, the collection is specified to be a collection of `Offenders` - only `Offenders` may be placed into the collection so only `Offenders` can be retrieved from it. The following will produce a compile time error:

```
// Causes compile time error
// List<Offender> can only contain Offenders and descendants of Offenders
offenders.add(new Officer());
```

Because the type of each object contained in the list is specified at compile time, no explicit cast is required when an object is retrieved:

```
// No explicit cast in Java 5 and above
// Cast taken care of by compiler so that type safety is ensured
Offender firstOffender = offenders.get(0);
```

- Each parameterized type also has an equivalent "raw" type resembling it's pre-parameterized (pre-Java 5) equivalent:

  ```
  // Legacy code – generates raw type warning in Java 5+
  List offenders = new ArrayList();
  ```

  This collection accepts elements of type `Object` and decedents of `Object` (all none-primitive types). An explicit cast is required when an element is taken out of this collection.

[1] Type information can be implicitly deduced by the compiler. For instance, in the case of the following code, the type of the reference object returned is exactly the same as the type of the single reference object passed:

```
public static <T> T convert(T t) { ... }
```

```
// Legacy code – explicit cast required with raw type
Offender firstOffender = (Offender)offenders.get(0);
```

With no compile time guarantee that only Offenders are stored in the collection, the above code incurs the risk of a runtime `ClassCastException` as shown in the following code:

```
// Broken, legacy code – throws ClassCastException on line 3
List offenders = new ArrayList();
offenders.add(new Officer()); // No compiler error pre Java 5
Offender firstOffender = (Offender)offenders.get(0); // Runtime error
```

This risk can be avoided if parameterized rather than raw types are used.

Further recommendation

Unchecked warnings often indicate a failure to take advantage of the possibility of stronger type safety from the use of a parameterized type. Fix unchecked warnings and only suppress them with good, documented reason. See “Resolve errors and warnings” for an explanation.

- The standard Hibernate implementation of the generic DAO ncluded in OMIS 3.0 is type safe according to the contract of the Hibernate documentation. See “Resolve errors and warnings” for an explanation.

Make parameters of method implementations final

Use the `final` modifier to prevent new values being assigned to the parameters of methods where a method is implemented (in all methods of concrete classes and none abstract methods of abstract classes). This modifier should not be applied to methods declarations in interfaces or abstract methods of abstract classes as it is redundant (it does not enforce that the parameter be final in implementing classes).

Prefer most general, base types to concrete implementations

If an API has multiple implementations and tiers in its inheritance hierarchy, use the most basic type furthest up in the inheritance hierarchy that provides the required functionality as parameter and return types. This is a method of hiding implementation details and allows the implementation to be switched should a better one be developed. For example, in the Java collections framework, prefer to use `List`, `Set`, `Map` over `ArrayList`, `HashSet` and `HashMap`. Use `Collection` when ordering and the possibility of repeat entries are not a concern.

For example, a `CourtCaseForm` may have a `List` of `ConvictionItems`:

```
public void setConvictionItems(List<ConvictionItem> convictionItems) { … }
public List<ConvictionItem> getConvictionItems() { … }
```

The implementation of `CourtCaseForm` with the property `convictionItems` of type `List<ConvictionItem>` can use any implementation of `List<T>` it wishes. What is more, if a new implementation of `List<T>` is developed and the currently used implementation is deprecated, the only changes that the developer of `CourtCaseForm` will be responsible for making will be in the `CourtCaseForm` implementation. The API and documentation will give no indication of which

implementation is used so clients will not be given any incentive to rely on specific features or behavior of the used implementation of `List<T>`. Any client that builds functionality based on the undocumented implementation of `List<T>` used will do so at their own risk.

It is also a good practice to adhere to this principal when declaring instance and local variables. For instance, in the code below, the reference variable of the local variable `aliases` is declared to be of type `List<T>` and not the actual `List<T>` implementation:

```
List<ConvictionItem> convictionItems = new ArrayList<ConvictionItem>();
…
courtCaseForm.setConvictionItems(convictionItems);
```

Prefer collections and maps to arrays

`Collections` (such as `Lists`, `Sets`) and `Maps` should be preferred to arrays. Collections and maps generally provide a richer API with more extensive functionality and can be manipulated with the Java Collections framework. The use of collections also allows a base type to be specified (`List`, `Set`, `Map`) and the implementation (`ArrayList`, `HashSet`, `HashMap`) to be hidden from the user of the API. Should a better (faster, for instance) implementation of the base type be introduced, the implementation used by a software module can be switched without each client of the API of the module having to alter code.

Recommendation

Use collections over arrays for exposed properties, return types, method arguments and internal implementation variables. In method bodies, prefer collections over arrays wherever possible.

Use the parameterized collection type over the raw type, specifying, exactly, the type of object the collection is to contain. Prefer using the most common interface type of the collection which gives all the functionality required by the client over more specific implementation types.

- See "Use parameterized types" and "Prefer more general, base types to implementation types".
- For collections of enums, use an `EnumSet`.
- Consider arrays "deprecated" and replaced by collections in OMIS 3.0.

Prefer `Lists`, `Sets` and `Maps` to `Vectors`

The Java `Vector` class is a pre-Collections (Java 1.2) growable array of objects. A `Vector`, unlike Collections classes, is always synchronized. A Vector has one standard implementation that is appropriate – that is itself. An inappropriate implementation is provided by `Stack`, which confuses the `Vector` inheritance hierarchy somewhat. Once an exposed property of an object or input parameter or output object of an API is declared as to be a `Vector`, only one appropriate standard implementation exists, a synchronized `Vector`.

Recommendation

Do not use Vectors. Use `Lists`, `Sets` and `Maps` as reference types. If a synchronized `List`, `Set` or `Map` is required, use the appropriate static method of the Collections utility class (eg, `synchronizedList`).

- Consider `Vector` and `Stack` "deprecated" in OMIS 3.0, use collections instead of `Vectors` and `Deques` with an appropriate implementation instead of `Stacks`.

Prefer `StringBuilder` to `StringBuffer`

Unless there is a compelling reason for a synchronized mutable string wrapper, use a `StringBuilder` instead of a `StringBuffer`. A `StringBuffer` is thread safe which is required if an instance of a mutable string wrapper is to be accessed across multiple threads. There should be no reason to share a mutable string wrapper across multiple threads as a mutable string wrapper should only appear as local variables in method implementations. `StringBuilder` is therefore the most appropriate choice of mutable string wrapper. Use this class for local string variables where an arbitrary number of large mutations are expected to be performed on the string. This will not incur the risk of a performance penalty as the use of the thread safe `StringBuffer` would.

- Also consider `String.format(…)` as a means of producing formatted text. Unless an arbitrary number of mutations is expected (say in a "for" loop or series of "if" statements), a mutable string wrapper is unlikely to yield any noticeable performance improvements.

Resolve compile time warnings and errors

Compared to compilers of other languages, the Java compiler delivers concise, insightful and well documented warnings often indicating possible bugs or overlooked logical outcomes. Attention should be paid to compiler warnings, their advice heeded to and concern resolved. The results should be that production code is warning free with use of the `@SuppressWarnings("…")` annotation kept to a well-documented minimum. In addition, a compiler error indicates that the current version of a module will not be included in the compiler output. Development and production code should be free of compile time errors.

Recommendation

Production ready Java source code should be free of compiler warnings and all committed code free of compile time errors. Use of `@SuppressWarnings("…")` is discouraged and should only be used when completely necessary. The justification for using `@SuppressWarnings("…")` should be indicated in an inline comment beside the annotation. Additionally, the `@SuppressWarnings("…")` annotation should be used at the most specific scope as possible. For example:

```
public List<Offender> findByCountryOfCitizenship(
            Country countryOfCitizenship) {
      @SuppressWarnings("unchecked") // API guarantees List of Offenders
      List<Offender> result = (List<Offender>)session.getNamedQuery(
            "findOffendersByCountryOfCitizenship")
            .setParameter("countryOfCitizenship", countryOfCitizenship).list();
      return result;
}
```

In the above, a named Hibernate Query Language query is retrieved and used to return a list of `Offenders` by `countryOfCitizenship`. Hibernate 3.x is compiled in Java 1.3 and therefore does not support generics. A cast is used in the Java 5+ project to convert the raw type List to the parameterized type `List<Offender>`. This would normally result in a compile time warning as the compiler has no way to guarantee that the runtime List returned by the Hibernate API will contain only instances of `Offender`. The warning would indicate that the cast is unchecked and therefore incurs the risk of a

`ClassCastException` at runtime. It is, however, safe to suppress this warning as the Hibernate API (for the list method) guarantees that, given the query used, a `List` exclusively of `Offender` objects is returned. (As Hibernate 3.x is compiled in Java 1.3 and does not support generics, this contract cannot be guaranteed in the method signature and is instead placed in the Hibernate documentation. A `ClassCastException` resulting from this code would indicate a violation of the Hibernate API's contract and not an error in the client code.) This is documented beside the warning suppression. Note also that the warning suppression is applied to the assignment. A reference variable needs to be declared to allow this as the annotation cannot be applied to a return statement. The warning suppression is not applied to the entire method as code may later be added to the method body which may generate unchecked warnings that would be unintentionally suppressed.

Avoid commenting out code

Code must never be commented out as a means of deactivating it.

Avoid deprecated Java library classes and methods

Java classes and methods which are outdated, superseded or contain bugs and should therefore no longer be used are annotated with `@Deprecated` and a "@deprecated" javadoc comment is added for the class or method declaration. For full details on deprecation in Java see:

- http://docs.oracle.com/javase/6/docs/technotes/guides/javadoc/deprecation/deprecation.html

Recommendation

Do not use deprecated classes or methods in Java. If a deprecated class or method is used, a solution that does not include deprecated code should be sought as a long term replacement. If a deprecated solution is used temporarily, the `@SuppressWarnings("deprecation")` annotation should be used with a comment explaining why the deprecated functionality was used and with what it is expected to be replaced. The comment should also warn of any potential side effects of using code with is acknowledged to be potentially harmful or inefficient.

Avoid overloading methods

Java allows too methods to have the same name providing their parameters do not match. For regular methods, overloading should be avoided.

Discussion

Method overloading can be considered a "historic" feature carried over from C++. What at first may seem a harmless, even helpful feature can introduce unclear logic, unexpected behavior and often indicates a failure to properly separated concerns. This is especially true when inheritance hierarchies are incorporated into a model design and polymorphic behavior is introduced.

Where the need arises for a class to have multiple overloaded constructors, consider instead using an appropriately named static factory method.

Avoid float and double types

The primitive float and double types as well as their wrapper Float and Double classes are imprecise when performing arithmetic according to the requirements of the 32-bit and 64-bit IEEE 754 standard for floating points. Do not use them!

Recommendation

For currency, use an integral type to represent the value in the lowest integral possible unit. For instance, use cents instead if dollars. Represent cents as a primitive integer in an API or as an instance of an Integer wrapper class for POJO properties (references to wrapper objects should be preferred in POJOs as the value of null can be represented).

If accuracy to a fraction of a currency's lowest possible unit is required – such as a tenth of a cent – use `java.math.BigDecimal`. For other decimal usages, also use `java.math.BigDecimal`.

- Presentation formatting should be handled in the presentation tier – presentation considerations should not be middle tier design considerations. A field type in the domain model should not be chosen according to the way in which it may or may not be presented in the presentation tier.
- It may be desirable – even necessary – to represent a property which is essentially numeric in its own wrapper class. For instance, price could be represented as a `DollarAmount` which is a concrete implementation of the abstract `MonetaryAmount`. The `DollarAmount` instance could track dollar amounts using an internal `java.math.BigDecimal` instance. In this case, the question must be asked as to how much functionality is gained from the added complexity and whether the extra complexity is worth the added functionality. In some cases, such as where normal decimal arithmetic does not apply, the benefit is clear:

```
offender.getHeight().changeInchesBy(10); // From 5'5" to 6'3"
if (offender1.getHeight().isGreaterThan(offender2.getHeight())) { … }
```

  In the case above (supposing the height property is of a `Height` type), everywhere a height in feet and inches exists, so does the ability to increase, decrease, compare, validate, etc, heights (assuming the Height class is used and implements this functionality). In other, quantitative value type, the argument for a wrapper class might not be as great:

```
offenderEmployment.setWageAmount(newWageAmount);
if (offenderEmployment1.getWageAmount()
      > offenderEmployment2.getWageAmount()) { … }
```

  In the above example, the monetary wage amount property is stored in cents as a `java.lang.Integer`.

Designing Inheritance Hierarchies

All reference types are to be interfaces. Instances of implementations of reference types are to be obtained via an instance producer (see Appendix <?>).

Separate types into interfaces and abstract implementations for simple inheritance hierarchies

The Java language does not allow a subclass to inherit from multiple parent classes. An attempt to partially compensate for the absence of this functionality is provided by interfaces that can inherit from

multiple parent interfaces. This permits multiple inheritances strictly in the realm of types, their contracts and type checking. This does not allow for the multiple inheritance of implementations.

Language background

Three types of type exist in the Java programming language; the following table gives a breakdown of type and its recommended usage in the case of simple inheritance hierarchies:

Name of type	Usage	Declaration Syntax	Naming convention
Interface	Contract – indicates properties and behavior which each instance of this type provides. Cannot be instantiated – wouldn't be useful for an instance of such an object to exist. Should always be used as a reference type.	`interface [NAME]` `interface [NAME] extends [OTHER_INTERFACE(S)]`	Name of a base type such as PersonName, Vehicle, Address, Person, Offender.

Abstract class	Common functionality – functionality that all instances of a type might use.
	Cannot be instantiated – wouldn't be useful for such an object to exist.
	Must not be used as a reference type. Never!
		`abstract class [NAME]`
		`abstract class [NAME] implements [INTERFACE(S)]`
		`abstract class [NAME] extends [BASE_CLASS]`
		`abstract class [NAME] extends [BASE_CLASS] implements [INTERFACE(S)]`
			Name of an abstract, base type preceded by the word Abstract, such as AbstractPersonName, AbstractVehicle, AbstractAddress, AbstractPerson

Concrete class, implementation	A type that is specific enough to be able to be instantiated into an actual object. This can be understood as a complete implementation of the contract of an interface/reference type.
	Should always be avoided as a reference type.
		`class [NAME]`
		`class [NAME] implements [INTERFACE]`
		`class [NAME] extends [BASE_CLASS]`
		`class [NAME] extends [BASE_CLASS] implements [INTERFACE]`
			Name of the type being implemented followed by Impl. PersonImpl, OffenderImpl, VehicleImpl, AliasNameImpl.

- Note that concrete classes can inherit for one another and there is often good reason for them to do so.

Recommendation

In the case of simple inheritance hierarchies where the requirements of all implementations are likely to be the same, separate the API of an object into an interface. The name of the interface should be the intended name of the type, such as PersonName. Of this type, there are three possible sub types:

ActualName, AliasName and MaidenName. A basic implementation of PersonName – with all the functionality required by all PersonNames, should be place in an abstract class called AbstractPersonName. Concrete implementations of the PersonName subtypes can then be derived from AbstractPersonName – such as ActualNameImpl which would implement ActualName, AliasNameImpl would implement AliasName, etc.

- Do not postfix the name of the interface with an indication that the interface is an interface (such as "Ifc"). The interface is the type – no other type should be used in by the client of the API.
- Equally, avoid indicating the type (interface or class) of type in the type's Javadoc comment.
- As a general rule, no code in a method body should contain types prefixed with Abstract or post fixed with Impl.
- It is permissible to call an abstract Person implementation AbstractPerson as no API should deal in AbstractPersons. The only time that AbstractPerson should be used other than in the declaration of AbstractPerson itself is when concrete implementations of Person derive from AbstractPerson for common Person functionality.

Delegate shared functionality to internal composited components for complex inheritance hierarchies

In more complex inheritance hierarchies, multiple inheritance of functionality might be required. As Java does not support multiple inheritance of classes (even abstract classes) the solution is to offload functionality into composited delegate classes that each instance of a subtype can instantiate (and use). The result is implementations that use common functionality shared by instantiatable types through the use of separate delegate classes.

Example

Assume that there exists an Event type. Event serves as a contract which guarantees all the properties and behavior that anything deserving to be considered of the type Event will provide. By extending Event, Java guarantees that an effort to honor this contract will at least be attempted by subtypes. Part of the contract is to detect conflicts of Events with overlapping dates if other resources are tied up in both of the Events. This is achieved by calling the method conflictsWith(Event).

Event serves as a base class for two further types: UserEvent and OffenderEvent. UserEvent has a user property: the user that scheduled the event; while OffenderEvent has an offender property: the offender about whom the event concerns (dental appointment, parole hearing, etc). UserEvent and OffenderEvent are level in the inheritance hierarchy – each inherits directly from Event and neither inherits from the other – there may be some UserEvent that require an offender and others that do not; likewise, there may be some OffenderEvents that require a scheduling user and some that do not.

An OffenderHealthAppointment is an Event which requires both a scheduling user and offender – it is of both OffenderEvent and UserEvent types. If an AbstractUserEvent provides a skeletal implementation of UserEvent functionality and an AbstractOffenderEvent provides a skeletal implementation of OffenderEvent, OffenderHealthAppointment will have to choose one skeletal implementation to inherit

from and manually implement the required functionality for the other type it extends (UserEvent or OffenderEvent). The skeletal implementation of the type not extended by OffenderHealthAppointment cannot be composited to prevent this duplication of code as additional unused functionality and properties from the composited skeletal implementation will be includes as part of OffenderHealthAppointment's graph. This would be extremely wasteful and should be avoided.

The solution is to offload the functionality required to implement UserEvent and OffenderEvent into separate delegate classes in a separate impl.delegate package (omis.<module-name>.domain.impl.delegate). These delegate classes are not part of the type hierarchy – they should implement neither UserEvent nor OffenderEvent – and should not be designed to cater to a specific type but rather provide common functionality that multiple types can use. OffenderEventDelegate should be thought of as providing common functionality to all Events which have an offender property. It should ***not*** be thought of as implementing OffenderEvent. It should not contain functionality common to all supertypes of OffenderEvent.

OffenderEventDelegate might look like the following:

```
package omis.schedule.domain.impl.delegate;
import omis.schedule.domain.OffenderEvent;
import java.io.Serializable;
public final class OffenderEventDelegate implements Serializable {
	private static final long serialVersionUID = 1L;
	public static interface PropertyGetter {
		Offender getOffender();
	}
	private final PropertyGetter propertyGetter;
	private Offender offender;
	public OffenderEventDelegate(PropertyGetter propertyGetter) {
		this.propertyGetter = propertyGetter;
	}
	public void setOffender(Offender offender) { this.offender = offender; }
	public Offender getOffender() { return offender; }
	public boolean conflictsWith(Event e) {
		if (e instanceof OffenderEvent) {
			return propertyGetter.getOffender()
				.equals(((OffenderEvent)e).getOffender());
		}
		return false;
	}
}
```

Notice that the delegate is in a separate package and is final. The delegate is not a required to follow standard POJO conventions but is required to be serializable. The delegate provides the offender property and has functionality to test whether or not an OffenderEvent conflicts with another Event. UserEventDelegate would be similar but with a user property. OffendeHealthAppointment – and any other type derived from UserEvent and/or OffenderEvent can leverage the shared functionality of the delegate class using the following idiom (only the use of OffenderEventDelegate is shown):

```
public class OffenderHealthAppointment implements UserEvent, OffenderEvent {
```

```
        private static final long serialVersionUID = 1L;
        private OffenderEventDelegate offenderEventDelegate =
               new OffenderEventDelegate(new OffenderEventDelegate.PropertyGetter() {
                      @Override public Offender getOffender() {
                             return OffenderHealthAppointment.this.getOffender();
                      }
               });
        …
        public OffenderHealthEvent(…, Offender offender, …) {
               …
               offenderEventDelegate.setOffender(offender);
        }
        public void setOffender(Offender offender) {
               offenderEventDelegate.setOffender(offender);
        }
        public Offender getOffender() { return offenderEventDelegate.getOffender(); }
        public boolean conflictsWith(Event event) {
               …
               if (offenderEventDelegate.conflictsWith(event)) {
                      return true;
               }
               …
        }
        public boolean equals(Object o) {
               …
               if (!getOffender().equals(that.getOffender()) { return false; }
               …
        }
}
```

- OffenderHealthAppointment does not have an internal offender field of its own – responsibility of maintaining this field is delegated to OffenderEventDelegate.
- The constructor and accessor methods are the only methods that access the delegate property (not additional functionality) directly. All other methods should use the property accessor method of the type itself. This is to allow the POJO to be used as a persistent entity with a property access strategy.
- Constructors and accessor methods can call methods on the delegate instance as the delegate type is declared final (in ability to extend the delegate removes the possibility of unexpected behavior been introduced into the constructor of the POJO).
- The conflictsWith(Event) method might call conflict detection methods from multiple types of delegate (UserEventDelegate for instance).
- In a similar manner to a POJO, the delegate does not read the properties delegated to it by the POJO directly (via the encapsulated member variable ***or*** its accessor method) other than in the property accessor methods. Instead, a custom object is used to access the properties via the POJO itself – which in turn will call the getter method of the delegate. The reason for this is that when a property access strategy is used in a persistent project, the getter of the property is often required to be called in order for lazy initialization to work. As this technique is used only in methods other than the properties accessor methods of the delegate, a recursive call to the property accessor methods of the delegate will not occur. The custom object is an

implementation of a delegate type's PropertyGetter (for want of a better name). The PropertyGetter is passed to the delegate on instantiation by the POJO and used in all methods other than property accessor methods (see the delegates implementation of conflictsWith(Event)).

- This solution will not work using a field access strategy by an ORM framework.

Functionality should be widened in derived types

A derived subtype should be able to do all that its super type can often with additional or more specific functionality. If less functionality is required in a subtype, the design of the inheritance hierarchy should be reconsidered.

Best practices for limiting functionality in derived types when unavoidable

If a limiting of functionality is required in a subtype derived from an existing, unalterable inheritance hierarchy, the fact that the functionality was taken out of the derived class should be documented and any attempt to use the functionality should result in an UnsupportedOperationException to indicate a logical, programming error.

For example, the Java collections framework allows unmodifiable "views" of collection types or maps to be returned. The static utility java.util.Collections class provides this functionality through its unmodifiableXxx() static methods. These methods return collections or maps with methods which allow the collection or map to be mutated are overridden to throw UnsupportedOperationExceptions. This is documented in the javadoc for these methods with little room left for ambiguity:

> *…attempts to modify the returned list, whether direct or via its iterator, result in an* `UnsupportedOperationException`*.*

Implementing equals/hashCode

- Be sure to override `equals(Object)` and not overload it with `equals(Offender)`. Use the `@Override` annotation on `equals` (and all other overridden) methods to reduce the risk of this error.
- If an object implements one of the `equals`/`hashCode` methods, it should implement the other as symmetrically as possible.
- In objects that implement `equals` and `hashCode`, there should be exactly one method with each of the names.
- In the case of equals, the method signature should be exactly:

```
public boolean equals(final Object obj)
```

- In the case of hashCode, the method signature should be exactly:

```
public int hashCode()
```

Do not use generated IDs in persistent entity class equals/hashCode

Many entity objects use a generated ID as a surrogate primary key. Such a key is generated when the object is first made persistent ("saved"). This incurs the possibility of an entity object been in a legal state as far as other properties go but not yet having a primary key (having not yet been made persistent). It is entirely reasonable to expect such an object to be compared for equality to other

objects or placed in a hash based collection. For this reason, the primary key of an entity object should not be used in its equals or hashCode methods.

Other entity objects use a unique provided "code" as a primary key. In such a case, an entity object is not considered legal if this property is missing. It may, in such a case, be permissible to use a non-generated primary key in implementing equals/hashCode.

Use the properties that make up an objects natural key in equals/hashCode

Decide which properties form an object's natural key and use these to implement equals/hashCode. A natural key composes of properties – perhaps mandatory – which distinguish one instance of an object from another of the same type. If a mandatory property that makes up an object's natural key is missing, is may be permissible to throw an IllegalStateException to indicate that no valid comparison can be made as not enough information is provided by an object to make a reasonable distinction between objects.

The properties which make up an object's natural key should be real world properties and not properties concerned with the storage and retrieval of the object in the persistent tier such as an entity's generated primary key.

Limit the use of lazily loaded properties in equals/hashCode unless completely necessary

In Java persistence, references to other entities as composited properties of entity object can be lazily loaded. This is achieved by the persistence provider providing a proxy object which, when first used loads the object graph of the property from the persistent storage. This is often used in one-to-many relationships.

When a property is lazily loaded, the data store lookup is delayed until the property is first used. The use of such a property in equals and hashCode might therefore cause a data store lookup the first time the property is accessed. In large collections of objects, this may result in multiple individual database lookups for each object in the collection.[2] This incurs the possibility of slowing down operations on the collection significantly.

- Note that this only applies to properties which are lazily loaded. It might be that ***all*** none-lazily loaded properties are loaded into a persistent entity object's graph the first time a none-lazily loaded property other than the ID of an object is used. In this case, it is permissible to use such properties in equals/hashCode as a persistent entity object's graph will most likely need to be loaded in order to perform a (useful) equal comparison or hash code generation.

Beware of recursively calling equals/hashCode on bidirectional composited classes

If two objects contain composited references to one another in a bidirectional relationship, it is possible that each may call equals or hashCode on its composited property. This will result in a recursive loop and

[2] That it is usually desirable to have read data available instantly prevents a series of similar lookup operations from been deferred and performed together in one database operation as record creations, updates or deletions might. Note that this is not the same as caching read information in the persistence context which might.

Java eventually throwing an exception. Care should be taken to ensure that equals and hashCode calls in bidirectionally composited objects do not result in recursive call by calling one another.

All POJOs

All POJOs should meet the requirements of the JavaBean specification. The link to the specification is given above in the Introduction section of this document.

All POJOs should implement Serializable

All POJOs should implement java.io.Serializable. For complex inheritance hierarchies, Serializable should be extended by the interface at the level below which all objects and interfaces will be persistent. All objects in a Serializable hierarchy must provide a serial version UID field to indicate the version of the object's serializable form. This will allow the version of the serializable form of the object to be tracked and allow the detection of incompatibilities between forms should the compiled bytecode of a class be updated on a running VM.

The standard for implementing Serializable in a simple inheritance hierarchy using abstract classes is similar to the following:

```
public interface Photo extends Serializable {}
public abstract class AbstractPhoto implements Photo {
	private static final long serialVersionUID = 1L;
}
public class OffenderPhoto extends AbstractPhoto {
	private static final long serialVersionUID = 1L;
}
```

- Fields marked with the private access modifier are not inherited and therefore cannot be overloaded. A serialization version UID should be declared for each in a serializable class inheritance hierarchy whether abstract or not.
- Delegates used in complex inheritance hierarchies should be serializable if they contain persistent properties. If the delegates contain functionality alone, or only transient properties – such as cached data – there is no need to have the delegate be serializable. If there is a chance that the delegate may store persistent properties in the future, the delegate should originally be serializable. In other words, a future version of a delegate should not become serializable if previous versions where not. If it is the case that a delegate that previously was not serializable acquires the need to become serializable, the original delegate should be left unmodified and a new delegate with a composition of the original delegate should be created with serializability. The new delegate should expose the functionality of the original through its API. A choice can then be made in POJOs of which delegate to use. The delegate used in a POJO can be easily switched if no indication of the delegate used is exposed through the POJO's exposed API (as they absolutely should not be).

Avoid generifying entity types

The property types of entities should be known at compile time thus eliminating the need for an entity type to be parameterized. If this is not the case the domain model should be refactored.

Always provide public default constructor in implementations

All POJO implementations must have a public default, no argument "nullary" constructor. If no constructor is declared, a default public, no argument "nullary" constructor is provided by the compiler. If other constructors are provided, the compiler will not provide the default and the programmer will have to declare one explicitly.

Discussion

When the persistence provider loads persistent objects from the data store, it instantiates new instances of the required entity object to store the data. If no default constructor is provided, the persistence provider will not be able to do this as the default constructor is used to instantiate new instances of the entity objects.

Recommendation

All POJOs should have a public default constructor provided explicitly by the programmer. This will prevent errors should the class later be updated to have a none-default constructor thus relieving the compiler of its responsibility to provide a default constructor should no other one be found.

Use property accessor naming convention for POJOs

As per the JavaBean specification, properties should be accessed by accessor methods. Two accessor methods are required, a getter and setter. The getter should be the property name prefixed with the verb "get" while the setter should be the property name prefixed with the verb "set". The method name should be camelCased. For instance, for the property `offenderName` of type `OffenderName`:

```
private OffenderName offenderName;
public void setOffenderName(final OffenderName offenderName) {
       this.offenderName = offenderName;
}
public OffenderName getOffenderName() {
       return this.offenderName;
}
```

Failure to conform to this specification resolves an API or framework of its contractual obligation to be able to perform expected operations with your POJOs if the API or framework is documented as requiring that POJOs used in the API or framework meet the JavaBean specification. Frameworks, such as Spring Forms/MVC and Hibernate and GUI/web front end builder tools use reflection to work with the properties of the objects with which they deal. If the JavaBeans naming convention is not adhered to such tools may have difficulty accessing the properties of POJOs with which they are expected to work.

In addition, POJOs should avoid methods beginning with the verbs get and set if they are not property accessor methods.

- POJOs should **not** provide accessor methods to work with properties as a different type to that exposed by their API. For instance, a POJO should not provide a `getDateAsString` or `setDateAsString` method for a property named `date` exposed as of type `Date`. Instead, formatting, validation and conversion should be handled in the presentation tier and should not be included in the POJO itself.

Group related properties into composited components

Individual properties should be grouped, as much as possibly, logically into separate composite classes ("components"). This should be done regardless of the table structures in which the values of the properties of the classes are to be stored. This separates concerns, encourages code reuse and provides a simpler, easier to use API for the parent POJO.

Do not call overridable methods in constructors of none-final classes

If a class is not final, it should not call methods which are not final in its constructor. This would allow the introduction of unexpected and undocumented behavior during the instantiation of an object. It would prevent an object from guaranteeing the contract of its constructor.

Recommendation

In cases where the "setter" constructor of an object and the setter method of a property perform a deep copy of the new property value, it might be tempting to do the following:

```
// Bad code – none final method of none final class called in constructor
public class UpdateSignature implements Serializable {
	private static final long serialVersionUID = 1L;
	// Other properties
	private Long date; // update date in milliseconds since 1 Jan 1970
	// Other constructors
	public UpdateSignature(UserAccount userAccount, Date date) {
		// Set other properties
		…
		setDate(date);
	}
	public void setDate(Date date)  {
		if (date != null) {
			this.date = date.getTime();
		} else {
			this.date = null;
		}
	}
}
```

A better technique might be to put the functionality for performing the deep copy of the new property value in its own private – and therefore not overridable – method. This method can then be called by the constructor and setter method:

```
// Good code – no none final methods called in constructor
public class UpdateSignature implements Serializable {
	private static final long serialVersionUID = 1L;
	// Other properties
	private Long date; // update date in milliseconds since 1 Jan 1970
	// Other constructors
	public UpdateSignature(UserAccount userAccount, Date date) {
		// Set other properties
		…
		internalSetDate(date);
	}
	public void setDate(Date date) {
```

```
            internalSetDate(date);
        }
        private void internalSetDate(Date date)  {
            if (date != null) {
                this.date = date.getTime();
            } else {
                this.date = null;
            }
        }
}
```

- A persistence provider might choose to proxy objects to allow functionality such as lazy initialization. For this reason, the getter and setter methods of a POJO's properties should not be final.
- Complex business logic should not be placed in a POJOs setter method. Validation and conversion should be performed in the presentation tier.

Use primitive wrapper classes instead of primitive types for POJO properties

Use the primitive wrapper types Integer, Long, Boolean, Short instead of the primitive types int, long, boolean, short. The primitive wrapper types allow an additional value – `null` – which can indicate that a property is not set. Using primitive types, a special value is required to indicate that a property is not set such as `-1`. This could lead to subtle and difficult to trace bugs as Java allows values to overflow (constant increments of an integer variable will eventually yield every possible value of integer – including the special value used to indicate an entity not yet persisted – such as `-1`). The default value of primitive wrapper types is null, which is a more appropriate default than that of its primitive equivalent (0 in the case of the primitive int type; false in the case of the primitive boolean type, for instance).

Recommendation

Use primitive wrapper types instead of primitive types for POJO properties this applies to surrogate primary key fields as much as it does business properties. Primitive wrapper types should be used even for properties which are specified as not been nullable in the persistence tier thus allowing the persistence framework a reliable means of assessing whether or not the property has actually been set.

Differences between primitive types and primitive wrapper types

Primitive wrapper objects and primitive variables can be used almost interchangeably in Java 6 (with the noted exceptions that all primitive wrapper types can hold an extra null value and that no implicit widening is performed on assigning a value to a primitive wrapper type). Arithmetic operators for Java primitive types are overloaded for their equivalent wrapper object type. Primitive types and their wrappers can be used in the same expression so that the following is valid and produces the expected value:

```
Integer one = 1;
int two = 2;
long three = one + two; // values are widened
```

Note that no explicit cast is required from the sum of the Integer and int (the type of this expression is an integral – can be assigned to either an Integer or int) to a long. The integral value is widened to the

type of the variable to which it is assigned as per the Java Language Specification. If a primitive wrapper type were to be used, an explicit cast is or some other form of conversion would be required (as of Java 6):

```
Long four = (long)one + two + 1; // explicit cast
Long five = new Long(one + two + 2); // call Long(int) constructor
```

When working with primitive wrapper types rather than primitive types, it is also important to note that two instances of the same primitive wrapper type that hold the same value do not have the same object identity but will pass an object equality comparison. For example, the following code snippet will run without throwing an assertion error:

```
Integer i1 = new Integer(2);
Integer i2 = new Integer(2);
assert i1 != i2;
assert i1.equals(i2);
```

Avoid C++ style "copy constructors", provide "setter" constructors with a consistent strategy for copying properties

In C++ a class's "copy constructor" accepts a reference of the same type as itself and instantiates a new object with the same property values as the reference. Uses cases requiring such a constructor in Java may indicate wastefulness in a design or implementation strategy and should therefore be rare. In a managed container, a copy constructor may also give the misleading impression that the replica object is managed too – which it most likely won't be as it is not instantiated through the container.

For C++ reference and pointer type data members, it is often unclear whether a shallow or deep copy of a variable should be made. For instance, if class Offender has a property named sex of type Sex, in the copy constructor it is unclear whether a shallow copy of the reference should be made:

```
// C++ code – cannot (should not) be transliterated
this->sex = that.sex;
```

Or whether a new object of type Sex should be instantiated and its properties set, thus providing a deep copy:

```
// More C++ code – cannot (should not) be transliterated
this->sex = new Sex(that.sex.code);
```

Although Java's simpler model for referencing variables goes part of the way in resolving this ambiguity, the requirements of a persistence provider's property access strategy and the general good practice of not invoking overridable methods in a constructor combine to further discourage the use of "general purpose" copy constructors. In a persistent POJO, the copy constructor in the above example would be transliterated using the following bad practice:

```
// Bad Java code - getSex() is overridable
// No place in a constructor but must be called
this.sex = that.getSex();
```

and:

```
// More bad Java code – see example above
this.sex = new Sex(that.getSex().getCode());
```

In a persistent POJO, a call to the property "getter" method may be required to load a lazily loaded property. This should be avoided in a constructor as the property method is overridable. The practice of providing a copy constructor should therefore be avoided altogether.

Recommendation

Provide two or three constructors as standard. The first should be a public, no argument constructor as indicated in the JavaBean specification. The second should accept and set all mandatory properties of the class which are not collections. If the class has properties which are not mandatory, a third constructor which accepts and sets all properties that are not collections – mandatory or otherwise - should be provided. Shallow copies of composited properties should be made.[3]

Properties which are collections – mandatory or otherwise - should not be included in any "setter" constructor as an attempt to copy them might invoke overidable code which may introduce unexpected behavior, break a constructor's contract, increase the time taken for an object to instantiate and introduce errors from an unexpected source that are difficult to debug. Instead, by excluding collections from a constructor, the user of a class is made responsible for managing the duplication of a collection.

If parts of an objects graph require duplication, provide a static utility method for that particular use case and document whether reference properties are shallowly or deeply copied. This is a much better strategy than a copy constructor as an entire object graph is not wastefully copies, extra functionality can be placed in the method specific to the use case allowing for code reusability and reference properties can be shallowly or deeply copied according to the use case.

- Use the open session in view pattern and implement serialization properly to allow serialized POJOs to be passed between application layers within the same transaction.
- For transactions propagated over a multi-request conversation, detach and reattach persistent entities.
- If a copy of an entire object's graph is required, consider reviewing the design of a process or the choice of implementation.

Summary

In practice, uses cases for providing a copy constructor in Java should be rare to none-existent. Alternatives exist and should be preferred which introduce benefits over a copy constructor. Where use cases exist for replicating large parts of an object's graph, the implementation should be profiled to identify potential performance bottlenecks.

---

[3] In persistent entities, this is appropriate for many-to-one relationships where a composited property value can be shared by multiple instances of an entity type. In the data tier, the foreign key of multiple records for each instance of the entity type will reference the same row in the table for the composited property type. Where the composited component's property values are stored in the same table as the parent entity, I am unsure as to whether or not the property values are copied to the new table if a shallow copy of the composited property is made. *I bet one is – SA.*

Avoid using the "clone" API

The "clone" API does not provide an effective means of dealing with an object's inheritance hierarchy. Its use is discouraged.

- A more efficient alternative to cloning an object's entire graph may be available, see the recommendation for avoiding using C++ style copy constructors.

Initialize single value properties to null (default) and collections properties to empty collections

Avoid setting default values when declaring the fields of single value POJO properties. Instead, for single value properties, the default value should be left as null (the default for instance variables). For properties of collection reference types, an empty instance of a collection implementation should be set on instantiation as the default value. For example:

```
// Good code – null defaults for single property values;
// empty collections instances for collection properties
public class CourtCaseForm implements Serializable {
       private static final long serialVersionUID = 1L;
       // Instance variable initialize to null by default
       private Judge judge;
       // Empty collection default for collection property
       // Property is set on instantiation
       private List<ConvictionItem> convictionItems =
              new ArrayList<ConvictionItems>();
}
```

- Local (method level) variables are not initialized by default. A compile time error will occur if a local variable is used without been initialized. Therefore, this standard does ***not*** apply to local variables. Local variable should only be initialized to null if the need for a null check is anticipated – otherwise by initializing a local variable to null, the compiler is deprived of the ability to notify the programmer of uninitialized local variable use via a compiler error. (This effectively changes what could be an uninitialized variable compiler error to a harder to debug and possibly conditional runtime `NullPointerException`.)

Exception/error handling

Exceptions should be thrown to indicate logic errors such as a violation of a business rule or an unrecoverable event such as a network failure or file system corruption. Business exceptions should not be used in the normal, anticipated flow of a business process to report information such as the state of an object. Rather, the condition where an exception might be thrown should be avoided preemptively with checks.

Use standard Java exceptions appropriately wherever possible

Java is packaged with exceptions catered towards specific needs. Some useful unchecked exceptions are listed below, read their documentation and use them. These exceptions should not be used to indicate reasonable outcomes of business processes but rather unexpected bugs:

- `IllegalArgumentException` – an illegal argument is passed to a method. The throwing of such an exception can be considered a bug in the overall application as illegal arguments should be caught and communicated to the used in the presentation tier.
- `IllegalStateException` – an object is in an illegal state – perhaps its property values are incompatible or incomplete. Use when an illegal state is detected. Do not use to prevent an illegal state. To prevent an illegal state surfacing in an object, use `IllegalArguemntExceptions` and checked business exceptions.
- `UnsupportedOperationException` – an attempt is made to perform and unsupported operation. This operation should be used for business logic errors/bugs and not operations in the business tier. To restrict usage of a method in the business tier, use checked business and permissions exceptions.

Throw business exceptions with concise but informative messages in business language

Checked business exceptions should indicate that the user has been allowed to attempt to violate a business rule. The language used in the informative message of the business exception should therefore be business language. For instance, if an attempt is made to save an offender name with a missing last name when a last name is required, the exception should read as follows:

> GOOD: *Offender last name required*

This is preferable to:

> BAD: *Offender last name is empty*

Or worse still:

> VERY BAD: *Offender last name is null*

The latter messages are simply statements of fact with no indicate of why the condition of such facts been true should be seen as an error. The latter two messages are worthless to the user – they at best imply what needs to be done to resolve the issue. The last example even uses technical language to describe what could reasonably be approached and resolved as a business issue.

Throw checked exceptions when an error is anticipated and recoverable

Checked exceptions should be thrown with the intention of been caught. A checked exception should therefore be anticipated and a strategy for recovery devised. Ideally, the client would check for conditions that might lead to the throwing of a checked exception and report them before attempting to call the method from which the checked exception would be thrown. A module API should provide the client with the means to do this. Thus, a client should never expose the user to a checked exception.

Recommendation

The Java language helps in the structuring of checked exception anticipation and recovery by requiring that a method that throws a checked exception declare it as part of its method signature:

- A checked exception should inherit from java.lang.Exception – either directly or indirectly.

- A checked exception ceases to be a checked exception when it inherits from java.lang.RuntimeException – either directly or indirectly. The exception then becomes an unchecked exception. See "Throw unchecked exceptions when an error represents an external and/or unrecoverable error".
- In the package omis.exception, two checked exceptions should be used either directly or as a base class for exceptions indicating an anticipated, recoverable error. BusinessException should indicate an attempt to violate an application's business rules. This might be a constraint defined in the requirements of a particular feature. A PermissionsException should be thrown when a user attempt to perform an operation for which they do not have the required security rights.
- A module API should provide the means to preemptively check whether – given the current state of related objects and position in the business process –an exception would be raised by continuation of the current business flow. This allows the client to prevent the exception from being thrown and from having to be recovered.

Anticipate exceptions and either prevent them or report more meaningful errors

Prefer:

```
// Good code
public int divide(int number, int by) {
	if (by == 0) {
		throw new IllegalArgumentException("Cannot divide by zero");
	}
	return number / by;
}
```

To:

```
// Bad code
public int divide(int number, int by) {
	try {
		return number / by;
	} catch (ArithmeticException ae) {
		throw new IllegalArgumentException("Cannot divide by zero");
	}
}
```

Or worse still:

```
// Very bad code
public int divide(int number, int by) {
	return number / by;
}
```

The first example anticipates that the client of the divide method might pass an invalid value, in this case zero, and reports the invalid input as an entirely appropriate IllegalArgumentException and thus avoids attempting the division. The second example relies on the division throwing an unchecked exception whenever an invalid argument is passed. This is inefficient as it involves a spawning an additional layer of exception handling. The actual process of the function is also attempted. This might not seem too significant in the case of simple, integral arithmetic. However, in more complex processes this could

needlessly instantiate large objects or open connections to relatively slow external resources. The final example is entirely bad as no attempt is made to catch an easily anticipated error.

The Java Tutorials provide the following guideline on which exceptions to throw when:

> If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.
>
> - *Unchecked Exceptions – The Controversy* (http://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html)

This guideline should be followed.

Never leave a catch block empty

Exceptions should always be dealt with:

- In business logic, an exception is never dealt with by printing its stack track. In business logic, it is unlikely that an exception will be dealt with by throwing another type of exceptions. An exception in business logic should be anticipated and dealt with by additional business logic. The end user should not be presented with a business logic error in the normal flow of an operation.
    - Rather than anticipate and catch a violation of business logic, client processes should operate in a way that prevents the violation from ever occurring.
- The handling of system exceptions will most likely be configured by the servlet container, application server or management framework. If an end user sees a system error, it should be attributable to a systems failure (such as a database error) or a bug.

Never throw Exception or Throwable, *never* declare a method to throw Exception or Throwable

Always throw a more specific exception pertaining to the type of error.

In business logic, *never* catch Exception or Throwable

Always catch more specific exceptions and deal with them in a way appropriate to the exception type.

Java Source Code Level Documentation

All exposed API elements should be documented with a javadoc comment. An exposed API element is one that is not declared with the private access modifier.

- For information on Java access modifiers, see "Controlling Access to Members of a Class" in The Java Tutorials (http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html).

Javadoc comments should document the contract of the elements of a module. The contract should state what a type represents and what its methods do. Javadoc comments should not include implementation details of a particular module – in other words, details of how a task is accomplished should be omitted. This is to allow implementations to be switched while honoring the stated API specification. Clients are therefore encouraged to use a software module according to a software module's contract and not what they expect its implementation to be.

- Javadoc comments begin with a forward slash followed directly (no whitespace) by two asterisk and are ended by an asterisk followed directly by a forward slash (`/** <comment> */`).

Document all exposed types, methods and data members

Javadoc comments should be placed on all exposed types and methods. For types, the documentation should specific what the type represents and, if possible, its intended usage. The first sentence should summarize this information. The type documentation should, at a minimum, include annotations specifying the author(s), the version – including the date that the type was last modified - and the module or application version since when the type was included. If the type is parameterized, information should be included using the parameter annotation. Optionally, related classes should be documented using the see javadoc annotation.

The following example gives minimal information for type:

```
/**
 * Schedulable event.
 *
 * <p>Schedulable events can be built into an event schedule.
 *
 * @author Stephen Abson
 * @version 0.1.0 (Jan 3, 2012)
 * @since OMIS 3.0
 */
```

The following example shows a comment for an abstract, parameterized class type that is part of a simply inheritance hierarchy of schedule builders for events:

```
/**
 * Builder for a recurring schedule.
 *
 * <p><b>All</b> recurring schedules are required to support configurations.
 *
 * @author Stephen Abson
 * @version 0.1.9 (Feb 10, 2012)
```

```
 * @since OMIS 3.0
 * @see DailyRecurringScheduleBuilder
 * @see WeeklyRecurringScheduleBuilder
 * @see MonthlyRecurringScheduleBuilder
 * @param C schedule configuration type
 */
public abstract class RecurringScheduleBuilder
      <C extends ScheduleConfiguration> extends ScheduleBuilder<C> { ... }
```

The comment includes details of related classes and the type parameter. Note that although this class is abstract, the prefix Abstract is omitted as this class might serve as a useful reference type (for instance, in cases where it is known that a schedule builder should be recurring but where the frequency of the recurrence is not yet know or determined). Thus, this class is considered part of the schedule builder type hierarchy; it is not simply the abstract implementation of a type. The documentation of the class indicates this.

Method documentation should state what the method does. The comment should be written in third person present simple with the first word of the comment been the verb that best describes the action of the method. For instance, the first sentence of the comment of a setter method might read as follows:

```
Sets the date range during which the event occurs.
```

The getter method for the same property might read:

```
Returns the date range during which the event occurs.
```

If the type of the property is mutable, a deep copy might be taken and returned by the setter and getter methods respectively. This should be documented:

Setter: `Sets the start date to a deep copy of the specified date.`
Getter: `Returns a deep copy of the start date.`

Method documentation should include descriptions of parameters accepted by the method, the value returned by the method and possible exceptions thrown (checked and unchecked) using their respective javadoc annotations. If it is possible that the method might return a null value, the conditions which might lead to such a result should be documented. The documentation of the exceptions possibly thrown by a method should begin with the word “if” and state the condition which would cause the exception to be thrown:

```
/**
 * Creates, persists and returns an offender health appointment schedule.
 *
 * @param userId user ID that scheduled appointment
 * @param appUserAttendanceRequired whether scheduling application
 * user is required to attend
 * @param offenderNumber offender number of patient
 * @param offenderAttendanceRequired whether offender is required to attend
 * @param scheduleBuilder builder to use to schedule event
 * @return newly created and persisted health appointment schedule
```

```
* @throws SchedulingConflictException if an attempt is made to schedule
* an event which conflicts with existing, scheduled events
*/
```

- There is no such javadoc annotation as date (eg, @date). Date information should be included after the source code file version.
- Javadoc supports some HTML markup directly. Paragraphs can be separated by the <p/> tag, code blocks can be placed in <code/> tags allowing sample usage of an API's feature and emphasis can be added to spans of text with the tag.
- The text that follows a javadoc annotation is considered part of that annotation until another annotation or the end of the comment is encountered. Newlines do not place comment content outside of the annotation.
- Data members should never be directly exposed. Instead, a class should provide public accessor methods to provide read and mutate private data members.

Place documentation in the type that it is first declared; inherit the comment in implementation classes

Unless additional functionality is added to a methods contract in decedent classes, write the documentation of a method once – in the type in which it is first declared – and inherit the comment from then on. For instance, a service interface might allow a DAO used by the service to be injected via a setter method:

```
/**
 * Sets the schedule data access object.
 *
 * @param scheduleDao schedule data access object
 */
void setScheduleDao(ScheduleDao scheduleDao);
```

In the service implementation, it will suffice to inherit this documentation:

```
/** {@inheritDoc} */
@Override
public void setScheduleDao(ScheduleDao scheduleDao) {
       this.scheduleDao = scheduleDao;
}
```

- No exposed API method, type or data member should be without a javadoc comment.
- The following provides insight into the javadoc conventions used while documenting the Java platform itself:
    - http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

Persistent Entities

Persistent entity POJOs

Prefer to deep copy properties which are of mutable types and are not persistent entities

Properties of types which are mutable but are not entities should preferably be deeply copied. The methods which set or return the mutable property should document whether a deep copy is made. This is to prevent the user of a persistent class from mistakenly believing that the following change to a mutable date will be persisted:

```
// Bad code – change to date will not be persisted
Date date = event.getDate();
date.setTime(3600000);
```

Recommendation

Each time the property of an immutable type which is not persistent is set or returned, a deep copy should be preferred. In some cases, this will allow the internal type that represents the property to be changed to an immutable representation. For example, a property of java.util.Date can be stored as a Long (not the primitive version of long as primitives are immutable):

```
private Long date;
public void setDate(final Date date) {
	if (date != null) {
		this.date = date.getTime();
	} else {
		this.date = null;
	}
}
public Date getDate() {
	if (date != null) {
		return new Date(date);
	} else {
		return null;
	}
}
```

- Changes made using a reference to a mapped composite component are persisted. In this case, a deep copy is not required. See “Use composition for properties”.

Implement equals/hashCode

The equals and hashCode methods of object should be overridden for each persistent object. These are essential in allowing the persistence provider to accurately manage the persistence content and ensure that objects are not overwritten or written multiple times. Follow the guidelines in “Implementing equals/hashCode”.

- The equals method should be overridden and not overloaded. The method should have exactly one argument of type Object and return a primitive boolean value. This is the signature of equals in Object. Use the @Override annotation to enforce the overriding of this and hashCode:

```
@Override
public boolean equals(Object obj) {
    …
}
```

Other than in getter/setter and constructor, use accessor methods for property access

Persistent entities may be runtime proxies of the original entity type. One of the reasons for this is to allow lazy initialization. Rather than simply return the field accessed by the property access method, the "getter" method of a proxied class may perform the actual data lookup on demand the first time the property is accessed. The method may then store and return the property's value. Similarly, the setter method may mark the persistent entity as "dirty" with changed needing to be persisted the next time the application context is flushed or committed. This allows all database operations to be performed at once and possibly combined for efficiency and shorter network delays.

Recommendation

In all a POJO's methods other than the constructors and property access ("getter"/"setter") methods, the property access methods should be used rather than direct field access.

The constructors should be excluded from this requirement as an object is most likely not persistent when the constructor is called. A managed container or persistence framework will most likely only call the default constructor either directly or as the inherited constructor of a dynamically generated proxy. At that point the entity will not be persisted (it will be in the process of been instantiated). Other constructors will most likely not be called by a managing container and therefore the entity will not be "managed" when the other constructor is invoked. Further, as a general best practice, an overridable method should not be invoked in a constructor - property accessor methods and entity classes cannot be final as this will prevent them from been proxied.

Property accessor ("getter"/"setter") methods should access fields directly as this is how the persistence provider will read and write the properties albeit with additional, aspect oriented functionality.

- When auto generating getters and setters in MyEclipse, be sure to select "keep field reference" for the "Field access in declaring type:"

If "use setter and getter" is selected, existing constructors will be modified to use the property accessor method and not direct field access. Direct field access is appropriate for a constructor so "keep field reference" should be selected.

Avoid storage type specific data types

POJOs should be designed so as to be independent of the storage type (database, flat file, etc) to which they are to be persisted. Storage type specific data types, such as java.sql.Date should be avoided.

Recommendation

- All dates should be implemented as Longs (wrapper and not primitive type so that null values can be represented) exposed as java.util.Dates. Where possible, a date and a time should be represented together as a single, exposed java.util.Date field.
- Where the above is not possible and a POJO has to represent a specific time in a specific day as separate time and day fields, discard irrelevant information in the setter and getter methods for each of the properties and document this behavior in each of the javadoc comments for the methods.
    - See otrack.offender.domain.OffenderLocationHistory in OMIS 3.0 for an example of how this might be implemented.
- No POJO source code files should contain any imports from the package java.sql. This includes java.sql.SQLException. See "Throw unchecked exceptions when an error represents an external and/or unrecoverable error."
- SQL blob types should be represented in POJOs as byte arrays. The property can be mapped as lazily initialized to imitate the advantages of the RDBMS managed SQL blob type.

Avoid formatter methods as read only properties

- Instead of "getEndDateAsString()" use JSTL format tags or equivalent to format output in the web tier.
- In POJOs, only property accessor methods should begin with "get" or "set" – failure to adhere to this convention may confuse web framework, GUI tools, etc
- Separates concerns – formatting, converting etc, belongs in presentation tier
    - Errors should be handled in appropriate tier

Entity mapping

Prefer to use property as opposed to field mapping

The default strategy that Hibernate uses to access properties is "property" access. This will call the getter and setter when reading from or writing to properties (when an object is persisted or retrieved). Alternatively, field access can be used whereby Hibernate uses reflection to read the private data members of the persistent entity.

The default strategy of property access should be used unless special circumstanced require field access. In the case of the latter, the reason for using field access should be documented.

Persistence Tier

Data access objects

Data access objects should contain fine grained functionality for accessing specific persistent entities. Each data access object should address a single persistent entity type.

Use standard DAO interface and Hibernate implementation

A standard interface and Hibernate implementation of a data access object is provided with the intention of allowing code reuse, standardizing data access and preventing code repetition for common functionality. The standard DAO assumes that the entity has a surrogate primary key property of type `Long`. The name of this property should be `id`. The standard DAO interface is:

- `omis.dao.GenericDao`

The standard Hibernate DAO implementation is:

- `omis.dao.impl.hibernate.GenericHibernateDaoImpl`

This implementation allows explicit name based configuration of persistent entities (rather than configuration derived implicitly from the type).

- DAO interfaces should be free of framework specific code. Framework specific code should be placed in the framework specific implementation of the DAO. For example, the interface `OffenderDao` should be implementable by `OffenderDaoHibernateImpl`, `OffenderDaoJpaImpl`, `OffenderDaoJdbcImpl`, etc. As `OffenderDao` is implementation framework agnostic, there should be no unsupported functionality in any of the implementation DAOs.

Recommendation

For the persistent entity `Person`, an interface for a data access object can be declared:

```
public interface PersonDao extends GenericDao<Person> { }
```

An Hibernate implementation can be defined:

```
public class PersonDaoHibernateImpl
      extends GenericHibernateEntityDaoImpl<Person>
      implements PersonDao { }
```

All basic persistence functionality such as looking a `Person` up by `id`, obtaining lists of persisted `Person` instances, persisting a `Person` instance and removing a persisted `Person` instance from the persistent storage is now provided through the Hibernate implementation of the `Person` data access object. The provided functionality should be specific to the type of the persistent object to whose persisted instances the data access object is providing access.

Place additional fine grained functionality in data access objects

Additional, fine grained, more general functionality can be added to the existing functionality provided by the standard interface and implementation of a Hibernate data access object. For instance, if a

method is required to produce a list of all persisted `OffenderCautions` with a current residence in a particular `Offender`, the functionality can be added as follows:

```
public interface OffenderCautionDao extends GenericDao<OffenderCaution> {
	List<OffenderCaution> findByOffender(Offender offender);
}

public class OffenderCautionDaoHibernateImpl
		extends GenericDaoHibernateImpl<OffenderCaution>
		implements PersonDao {

	private static final String
		FIND_BY_OFFENDER_QUERY_NAME =
			“findOffenderCautionsByOffender”;

	private static final String OFFENDER_PARAM_NAME = “offender”;

	@Override
	public List<Person> findByOffender(final Offender offender){
		@SuppressWarnings(“unchecked”)
		List<Person> cautions = (List<Person>) getSessionFactory()
			.getCurrentSession().getNamedQuery(
				FIND_BYOFFENDER_QUERY_NAME)
			.setParameter(OFFENDER_PARAM_NAME, offender)
			.list();
		return cautions;
	}
}
```

Notice how the API for the `OffenderCaution` data access object accepts an instance of a persistent type used in the lookup criteria. Each data access object should only be required to perform data access for the specific entity type. A data access object for a single, specific entity type should not perform lookups, removals or other data manipulation for other entity types. Entity object required by the data access object of other types should be provided by the client.

Use named HQL queries (in Hibernate)

Hibernate allows HQL queries to be named and placed in a Hibernate mapping file. This practice should be preferred to hard coding HQL queries into the source code of DAO implementations. This allows named queries to be modified without recompiling Java source code files (which would otherwise contain the HQL query) and allows the named queried to be checked for syntax and mapping errors when the Hibernate session factory is initialized.

Example

The named HQL query can be placed in the Hibernate mapping file using the `<query/>` tag. The query tag should follow any class mappings:

```
<query name="findOffenderCautionsByOffender"><![CDATA[
	select caution
	from omis.caution.domain.OffenderCaution caution
	where caution.offender = :offender
	order by caution.dateRange.startDate, caution.dateRange.endDate,
```

```
                caution.source.name, caution.description.name
]]></query>
```

See the example above ("Place additional fine grained functionality in data access objects" in the body of the implementation of `findOffenderCautionByOffender`) for how this query might be used.

- Specify that the body of the `<query/>` tag is character data using a XML CDATA (`<![CDATA[]]>`) section so that the content is not parsed as XML.
- Always order queries that could possibly return more than one instance of an entity.

Use constants instead of string literals for query names and parameters

Rather than hard coding the names of queries and parameters in DAO implementations, use constants defined at the beginning of the class. Constant names should be entirely upper case with underscores separating words. The name of a named query constant should reflect the name of the named query without target entity information (this can be inferred from the DAO) and with the suffix `_QUERY_NAME`. Constant names for the parameter should mirror the name of the parameter with the suffix of `_PARAM_NAME`. See the example above ("Place additional fine grained functionality in data access objects") for an example of constants for query names (`FIND_BY_OFFENDER_QUERY_NAME`) and parameters (`OFFENDER_PARAM_NAME`).

Avoid native SQL queries – prefer report objects populated used HQL (in Hibernate)

Native SQL queries reduce the portability of the application and must be avoided. The querying language of the persistence provider (HQL, JPA QL, etc) should be used for queries and reporting instead.

- Report objects might be useful for none-POJO report data.

Transaction demarcation (in Spring)

Transactions should be configured declarative by the use of Spring's AOP pointcut with a transaction advisor. The pointcut should be applied to the web module for which a transaction is required. The following shows the pointcut being applied to the controller for alerts:

```
<aop:config>
    <aop:pointcut id="alertControllerOperation"
                  expression="execution(* omis.alert.web.controller.AlertController.*(..))"/>
    <aop:advisor advice-ref="transactionAdvice"
                 pointcut-ref="alertControllerOperation"/>
</aop:config>
```

A session will be propagated across the entirety of a web request by Spring's `OpenSessionInViewInterceptor`. The client should not, however, rely on the interceptor to commit database transactions (even if the flush mode is set to do this). Web modules that require transactions to be committed should do so by declaring themselves transactional.

- ***Do not*** put framework (Spring, Hibernate, etc) specific configuration in DAO APIs, service APIs, service implementations or domain models. In DAO implementations, the use of the API of the persistence provider will be required. However, transaction or security related configuration

specific to a framework or management container should not be included in DAO implementations.

Exceptions causing rollback of a transaction

All unchecked (derived directly or indirectly from `RuntimeException`) and checked business exceptions (derived directly or indirectly form `BusinessException`) will automatically cause the rollback of a transactional operation declared with Spring's AOP pointcut. Services should not throw exceptions of types other than these.

Service layer

The service layer allows a business process to be performed. The service layer should provide a simple API which allows clients to perform business logic and persist business objects. Service objects should be the only managed components injected into the client tier. The service layer should provide the client with all it needs to perform a business operation without need of a data access object or direct access to the persistence tier through other means (such as an injected session factory). Service objects should have an interface which takes the name of the service object (without a prefix indicating the type of object) and an implementation (typically the service object name post fixed with Impl). For instance:

`UserAccountManager` – service itself, interface
`UserAccountManagerImpl` – service implementation, object extends interface
`VictimRegistrationService` – service itself, interface
`VictimRegistrationServiceImpl` – service implementation, object extends interface

Only the interfaces themselves should be used in client code; managed implementations should be injected by using reference variables of the interface types by the IoC container (Spring, EJB, etc). For instance, in the case of a Spring web module with Spring autowired dependency injection:

```
@Controller
@RequestMapping(“/victim”)
public class VictimRegistrationController {
	@Autowired
	private VictimRegistrationService victimRegistrationService;

	@RequestMapping(“/register.html”)
	public ModelAndView registerVictim(Victim victim)
			throws VictimRegisteredException {
		victimRegistrationService.registerVictim(victim);
	}
}
```

- Only the interface is used by the client – the container (Spring) manages the injection of the implementation according the reference type (the interface).
- The service performs business logic (check to see if the specified victim is already registered) and throws a checked exception declared to force a rollback of the active transaction if business rules are violated (if the victim is already registered).
- The service provides the web module with all it needs to perform the required operation. The client does not need to know which DAOs are involved.
- Client code is easy to understand and contains only client concerns.

Keep service objects free of framework specific (Hibernate or Spring) code

Services and their implementations should be framework agnostic and therefore be able to function, without modification if a framework is switched or dropped.

- Data access must be accomplished through injected DAOs. There should be no direct data access in service layer components.

- Transaction demarcation should not be determined at the service layer level. In most cases, the most appropriate place for transaction demarcation is the client tier (such as web request level).
- Security should not be determined at the servicer layer level. In most cases, the most appropriate place for security is in the client tier (such as web request level).
- Dependency injections must be performed by Spring application context XML file and ***never*** Spring specific autowiring annotations as this would tie the service implementation to framework specific dependency injection.

Service methods

The conventions for naming service methods are as follows:

- `T create(properties…)` – creates a new instance of an entity, returns the created entity.
- `T update(T entity, properties…) -` updates an existing instance of an entity, returns the updated entity.
- `void remove(T t)` – removes an instance of a persisted entity.
- `List<T> findAll()` – returns all persisted entities. If the entities can be invalidated (have a `valid` property), this method will only return valid entities. If the entities can be sorted (have a `sortOrder` property), the entities will be ordered according to their sort order. Not recommended where `T` extends `OffenderAssociable`.
- `List<T> findByOffender(Offender offender)` – for entities with a many-to-one relationship with offender, return the entities associated with the specified offender. `T` should extend `OffenderAssociable`.
- `T findByOffender(Offender offender)` – for entities with a one-to-one relationship with offender, return the entity associated with the specified offender. `T` should extend `OffenderAssociable`.

JSTL/Spring MVC Web Tier

Spring Web Controllers

A Spring controller should provide the ability to perform one or more related business operations.

Controllers must be named according to the related operations of which they allow the performance

Controller names should reflect the business operations that the controller is to provide the ability to perform. Often, this will be the name of the module itself:

- `CautionController`, `OffenderSearchController`, `SecurityController`

In larger modules having business operations that can be grouped according to business objects with which they operate, separate controllers maybe desirable to separate operations according to the business objects:

- `HealthIssueController`, `UserGroupAdminController`, `EmployerController`

Inject services for work with business objects

Services can be autowired into controllers for access to the service tier. Data access objects must not be injected into controllers for direct access to the persistence tier.

- Never inject a data access object into a controller.

Use open session in view pattern to prevent lazy initialization exceptions

The open session in view interceptor should be configured for Spring MVC application wide. This interceptor will tie a lazily initialized persistence context to the thread executed for the duration of the web request. Transaction management can then be demarcated at the controller level using the Spring `@Transactional` annotation. This should prevent lazy initialization exceptions.

Use Property Editors to Resolved Identifiers to Business Object Instances

Property editors must be used to lookup business object using the service layer by their unique identifier (ID). Business object instances should not be looked up manually in the controller method.

- Property editor factories should be used to provide property editor instances for business object types. Property editor factories can be configured to be autowired into the controller for which its use is required. See Appendix <APPENDIX> for information on property editor factory design, implementation, and configuration.

Adhere to a URL Pattern and View Naming Convention

See Appendix <APPENDIX> for standards on URL patterns, view file and folder naming conventions. To these standards, adherence must be made.

Prefer the return type of ModelAndView over string from controller methods

For consistent method signatures in controllers, it is preferable to return a ModelAndView over a string from a controller method. A redirect from a controller method via a ModelAndView can be achieved using a view name similar to the following:

```
return new ModelAndView(INDEX_REDIRECT_VIEW_NAME);
```

Runtime Exceptions must never be caught in web tier components

A runtime exception should never be caught in a web tier component. Instead, the full stack trace of the exception should be presented to the user.

Discussion

Runtime exceptions represent errors that are not anticipated by the programmer such as persistence tier mapping errors, class cast exceptions, database or other hardware failures, and memory or disk space shortages. Such an error is not expected to be recovered from. Thus, the full stack trace of the exception should be displayed on such an error being encountered. The full stack trace of this bug or networking/hardware failure can then be reported to the application support team.

Business rule violations should be anticipated and an attempt should be made by the application to recover from them.

Use constants for model keys and view names

Rather than hard coding literal string values as model keys (including command objects) and view names, use constants defined at the beginning of the controller. The name of the constant should be in upper case with each word separated by an underscore. In the case of models keys, the name of the constant should be the key value with the suffix of `_MODEL_KEY`. For instance, the model key for a user account form ("userAccountForm") would be declared as:

```
private static final String USER_ACCOUNT_FORM_MODEL_KEY
            = "userAccountForm";
```

In the case of view names, each view name should have the suffix of `_VIEW_NAME`. For instance:

```
private static final String USER_ACCOUNT_LIST_VIEW_NAME
            = "user/admin/userAccount/list";
```

If a view name performs a redirect (is prefixed with `"redirect:"`), the suffix should be `_REDIRECT`, for instance:

```
private static final String USER_ROLE_LIST_REDIRECT
            = "redirect:list.html";
```

If the redirect includes URL parameters, the constant name should reflect this by indicating which parameters are included and how they are used, for instance:

```
private static final String USER_ACCOUNT_LIST_BY_USER_REDIRECT
            = "redirect:list.html?user=%d";
```

The controller method that used this redirect view name would set the value of the user query parameter using `String.format(…)`. For example:

```
return new ModelAndView(String.format(
                    USER_ACCOUNT_LIST_BY_USER_REDIRECT, user.getId()));
```

This is in preferences to concatenating strings.

Another example of a view name constant with a redirect that indicates the URL parameters is:

```
private static final String ENTER_USER_ACCOUNT_WITH_REDIRECT_URL_REDIRECT
            = "redirect:/user/userAccount/enterUserAccount.html?redirectUrl="
                        + "/user/admin/userAccount/edit.html";
```

JSPs

Never hardcode option labels and values into a JSP

Form element values and label must never be hard coded into options form HTML form components such as – but not including – checkboxes, radio buttons and drop down list items. Instead, these values and representative labels should be taken from appropriate middle tier components such as business objects persisted to tables or Java enumerations.

Never hardcore messages into a JSP

For static text such as –but not limited to – titles, headers, labels, button labels, controller labels and help messages, the message content should never be hard coded into the source of the JSP. Instead, localization (internationalization, i18n) should be used.

- See Appendix <APPENDIX> for information on how to use localization.

Spring Forms

For multiple choice components, represent not set with an empty string

Assuming “Time Block” is a field on a form represented by the timeBlock property of the command object of type TimeBlock. A drop down is to appear on the form containing a list of possible time blocks from which the value of the property is to be set. The following code snippet shows the markup for such a field:

```
<form:label path="timeBlock">
      <fmt:message key="timeBlockLabel"/>
</form:label>
<form:select path="timeBlock">
      <form:option value="" label="..."/>
      <form:options items="${timeBlocks}"itemLabel="label"
                  valueLabel="value"/>
</form:select>
<form:errors path="timeBlock"/>
```

The resulting HTML will display, as the first drop down list item a zero length “empty” string value represented by the label “…”. It should be assumed that the property editor will resolve a null or empty

string value passed as a URL parameter to null. Thus, this value represents the fields as having not been set. For the result of the options presented in the drop down, their “label” property will be displayed as a representation of the value of their “value” property.

- ***Never*** use other values such as the number zero or the string literal “null” to represent an unset field.
- It should be assumed that the property editor will treat a null value or zero length string URL parameter as a null.

Testing

Java Testing

Test code should be exemplary

- Example usage
- Write tests against an API's documentation and not implementation code
- view implementation when tests fail, fix them and check comments are still valid, if not, reconsider design

Java Persistence Testing

Test persistence operations by flushing and clearing the persistence context

That changes to a persistent entity - proxied or not - can be made, or that an entity can be persisted by calling makePersistent in a test does not indicate that an entity will be in a valid state for persisting to an actual data store in production. To verify this, a test should flush the persistence context. The commits all changes to the current transaction which is usually rolled back once the test finishes.

Spring/Hibernate Recommendation

The persistence context in Hibernate is flushed through the Session object's API. The current session object – usually thread specific – can be obtained through the Hibernate SessionFactory. A SessionFactory is usually configured and retrieved in a project's Spring application context.

- See relevant appendix B.

Appendix A: Useful tools

Use FindBugs for more extensive bug detections and warnings

FindBugs is a Java tool which scans Java source code or compiled byte code for possible errors and suggests improvements. Errors detected include logical errors, initialized but unutilized resources, redundant code, confusing and misleading names and unexpected behavior. FindBugs comes as a command line tool or an Eclipse plugin. Possible bugs are detected and displayed in Eclipse in a similar fashion to Java warnings.

Recommendation

Periodically run code through the FindBugs Eclipse plugin. FindBugs can be downloaded from:

- http://findbugs.sourceforge.net/

Appendix B: Testing the Middle Tier with TestNG

Tests should be placed in a package of the src/test/java folder of the OMIS 3.0 project. Test classes (and test helper classes) placed in this folder will not be exported to production builds. Each test class should be specific to a particular layer of the middle tier and should be placed in a package named appropriately. The last (right most) package in the package's path should be `testng` to allow for other testing frameworks to be used. The rest of the package (left part preceding testng) should be identical to the package of the layer of production middle tier code been tested. For instance, if testing a data access object in the Offender module, the package used for TestNG test classes should be:

`omis.offender.dao.testng` (in the src/test/java folder)

For testing a domain object, in the same module, the package used for TestNG classes should be:

`omis.offender.domain.testng` (in the src/test/java folder)

Test classes should be named according to the class of which the functionality is to be tested. For simpler classes, the name might simply be the name of the class been tested post fixed with `Tests`. For instance, the test class for a weight domain object in the demographics module might be named:

`omis.demographics.domain.testng.WeightTests` (in the src/test/java folder)

For domain objects with more complex functionality, specific functionality might be broken out into separate test classes. For instance, a test class might test the object equality of the Offender domain object. This class might be called:

`omis.offender.domain.OffenderObjectEqualityTests` (in the src/test/java folder)

This test class would essentially test the `equals(Object)` and `hashCode()` methods of the Offender entity object.

TestNG test cases can be declared by using the `@Test` annotation at the class or method level of a test class. Tests are performed by evaluating a series of Java 5 `assert` statements.

Obtaining a Spring Persistence Context and Testing the Hibernate Persistence Tier

Tests of managed Spring beans and Hibernate persistence operations can be performed using the Spring TestNG test library. This allows the retrieval of a Spring application context, managed bean autowiring and transaction management/automatic rollback. In order to use Spring's Hibernate TestNG test library, a test class must:

- Provide the location of the persistence context using annotation metadata:

```
@ContextConfiguration(
        locations = "file:src/main/webapp/WEB-INF/applicationContext.xml",
        loader = GenericXmlContextLoader.class
)
```

- Extend `AbstractTransactionalTestNGSpringContextTests`.

Alternatively, an abstract class with this functionality will be provided in the OMIS 3.0 project. Extend this class for the above functionality:

- `omis.testng.AbstractTransactionalTestNGSpringOmisContextTests`

By using either method, Spring managed beans such as services and DAOs can be injected into the tests using the `@Autowired` annotation as would be done in a Spring web controller:

```
@Autowired
private OffenderService offenderService;
```

Each test method will implicitly begin and rollback a Hibernate transaction with its own persistence context.

Testing Transactional Hibernate Operations

- The following describes how to flush and clear the persistence context in Spring/Hibernate TestNG tests. Neither operations should ever be performed in production code. Instead, Spring transaction management in Spring web controllers should be used to control the persistence tier (never in services or DAOs thus ensuring that the service or DAO is not tied specifically to the Spring framework).

In order to properly test persistence operations– especially ones involving creates, updates and deletes – it will almost always be beneficial to flush and clear the persistence context and reread the persisted data. Using the Spring/Hibernate TestNG framework, the persistence context will be flushed to the current transaction – which will be rolled back once the test is finished causing no changes to actually be written to the database.

An autowired Hibernate SessionFactory is required to test the persisting of transaction operations:

```
@Autowired
private SessionFactory sessionFactory;
```

Within the test itself, data modification operations can be performed. Assuming an autowired `OffenderService` is been used to test the saving of a new offender:

```
offender.setSex(newSex);
…
offenderService.save(offender);
```

If the test was to finish at this point, no actual database operation would be performed. There would be no indication that the Hibernate mapping matched the associated table structure of the Offender class. Nor would there be any verification that the data that was saved is the data that will later be retrieved. In order to confirm that database operations are performed correctly, the persistence context must be flushed:

```
sessionFactory.getCurrentSession().flush();
```

This will result in an SQL INSERT statement for the offender. What is more, the auto-generate primary key of the Offender – in this case offender number – would be determined (from an Oracle sequence or MySQL auto-increment, for example) and assigned to the new record. This key can be retrieved:

```
Integer newOffenderNumber = offender.getOffenderNumber();
```

At this point, the persisted offender still exists in Hibernate's persistence context. A lookup by the new offenderNumber would not cause the data to be reread from the database but rather for the same offender object to be returned as was just persisted (albeit with a newly generated offender number that all other references to the object would now have). In order to get fresh data from the database, the persistence context must be cleared:

```
sessionFactory.getCurrentSession().clear();
```

No objects will now be managed in Hibernate's persistence context. When the same offender that was saved is later looked up during the same session, Hibernate will check the persistence context and on discovering that no object with the specified offender number is currently managed will perform a database lookup:

```
offender = offenderService.findByOffenderNumber(newOffenderNumber);
```

The above statement will cause a SELECT statement to be generated and executed. From the returned Offender, the persisted properties can be tested to make sure that they were correctly saved:

```
assert offender.getSex().equals(newSex)
            : "Wrong sex persisted; expected: " + newSex
            + "; found: " + offender.getSex();
```

- Never perform flush or clear operations directly in production code. Use Spring transaction management in the web module that performs the data access.

Using the MyEclipse TestNG Plugin

The TestNG plugin can be added to a MyEclipse configuration in the MyEclipse Configuration Center. A whole test class can be run by navigating to the class and selecting Run -> Run As -> TestNG Test from the menu. Specific methods of a test class can be run by highlighting the name of the specific method and selecting the same menu item.

Appendix C: Domain Model Best Practices Quick Reference

Note: all code given as examples in this appendix is good – albeit compressed - code. No examples of bad code are given.

- Separate the domain model into API interfaces and concrete implementation classes.
- Place the API interface reference types in the `domain` package of the module. The content of this package should be considered the domain model. The interfaces within this package should always be used as reference types (even within implementation classes). The name of this type should reflect the real word object that the type represents. There should be no technical prefix or postfix to the name of the type (no `Ifc`).
- Place the implementation and abstract classes in (`domain.impl`). These classes should ***never*** be referenced in ***any*** Java code. Prefix abstract implementations with Abstract; postfix concrete implementations with Impl.
- Make all domain objects serializable, provide a serial version UID in every implementation or abstract class that implements the domain object type:

```
public interface Name extends Serializable {
}
public class NameImpl implements Name {
      private static final long serialVersionUID = 1L;
}
```

- Provide an explicit public, default, no argument constructor. This constructor does not have to do anything.
- Do not call overridable methods in a constructor:
- For properties, hide all fields by making them private, provide "camelCased" public accessor methods:

```
private Name name;
@Override
public void setName(Name name) { this.name = name; }
@Override
public Name getName() { return name; }
```

- If a property is to be persisted, it must be of a serializable type.
- In the implementation class, use the @Override annotation on all methods that are part of the reference type API.
- Use primitive wrapper types instead of primitive types for properties as object wrapper references can store null values:

```
private Integer amount;
```

  All primitive wrapper types are serializable, they can be used to store null values (primitive types cannot and would require a special value to represent unset, which is undesirable).
- Avoid these types for properties: `int`, `long`, `boolean`, `char`, `float`, `double`, `Float`, `Double`, **`java.sql.`**`Date`, `java.sql.Time`, `java.sql.Timestamp`.
- Prefer these types for properties: `Integer`, `Long`, `Boolean`, `Character`, `BigDecimal`, **`java.util.`**`Date`.

- Deep copy mutable properties which are not mapped persistent entities or mapped composite components (such as properties of type java.util.Date).
- Collection properties should be `Sets` and not `Lists`, for multiple reasons:
    - Semantics – a person can be said to have a set of aliases by which they are known. A person doesn't really have a list.
    - Sets do not allow duplicates.
    - Although lists keep track of order, the order in which elements of a collection property of a business object are returned should not be considered part of the API of a business object. Business tier operations should not rely on collections properties being in a certain order. (This is most likely a presentation concern to be resolved in the presentation tier; often the querying feature of an ORM framework can supply the presentation tier with an ordered list).
- Initialize collection properties to `HashSets` in the implementation (always use `Set` as the reference type for both instance variable and accessor methods):

```
private Set<Name> names = new HashSet<Name>();
```

- The names of properties which are collections should be pluralized.

```
public void setNames(Set<Name> names) { this.names = names; }
public Set<Name> getNames() { return names; }
```

- Provide methods which allow the manipulation of a collection's elements. This forms part of an object's business logic. Allow for usage such as:

```
person.addName(newName);
person.removeName(oldName);
user.grant(adminRole); // Add admin role to user's roles set
```

- In the implementation of the business logic of a class, but not in constructors or in a specific property's accessor methods, do not refer to a property using its private field. Instead use the accessor method:

```
public void addName(Name name) { this.getNames().add(name); }
```

- Place business logic in classes which semantically defines one objects relation to another:

```
public boolean conflictsWith(Event event) { … }
… below is part of another methods body …
if (proposedEvent.conflictsWith(scheduledEvent)) { … }
```

- Implement both `equals(Object)` and `hashCode()` using the `@Override` annotation to ensure that the methods are overridden and not overloaded. In each method, use each property's accessor methods and not the property field, for instance:

```
@Override public equals(Object obj) { … }
@Override public int hashCode() { … }
```

- Check for recursive method calls in `equals(Object)` and `hashCode()`.
- If a required property is null when equals or `hashCode()` is called, throw an `IllegalStateException` to indicate that not enough information about the object is available to adequately perform the operation.
- Neither the implementation class nor the public accessor methods of persisted fields should be declared final. Declaring them final would prevent proxying in managed environments (such as Spring and/or Hibernate).

- Choose names wisely – avoid names which conflict with core Java classes or classes in other packages of the application.
- When convention is broken – document why.
- Document all none-private classes, methods, properties and fields using javadoc comments.
- Include in javadoc comments:
    - A description of the purpose of a class or what the class is intended to represent.
    - The purpose of a method and any side effects.
    - Conditions which are expected to be true once a method is invoked.
    - Details of what a property is intended to represent.
    - A description of the parameters and returned value of the method.
    - Insight into the exceptions which might be thrown during the invocation of a method.
    - Exemplary code showing usage of the class or method in a <code/> block.
- Do not include implementation details in a javadoc comment. For instance, if a method returns `List<T>`, do not indicate the implementation type of the list (`ArrayList`, `LinkedList`, etc). As another example, if a property is exposed as a `Date` but stored as a `Long`, do not document this. Rather, it will suffice to note that a deep copy of the date is made when the property is accessed.

Appendix D: Creating persistent entities and mapping with Hibernate

Writing the POJO

Choosing name, package and type (of type)

- Name should always be a noun
- Could be a compound noun
- Some collective nouns are permissible
- Should never be pluralized
- Avoid postfixes which indicate type of type (avoid Ifc, etc)

POJO Requirements

- Implement serializable - serialVersionUID
- Private camelCased field with public camelCased accessor methods make a property

Defining relationships with other POJOs

- Composited component property or collection of components

Making the POJO A Persistent Entity

- Equals/hashCode
- Internal property reference
- No indication that the POJO is a persistent entity other than an unnatural surrogate primary key

"Is a" versus "Has a" versus "Delegate"

- "Is a" – impacts type hierarchy
- "Has a" – defines association between POJOs
- Delegate is hidden – no impact on type hierarchy or exposed association

Writing Mapping File

Defining relationships with other persistent entities

Appendix E: Basic equals/hashCode javadoc Comments for Persistent Entities

For the `equals(Object)` method of persistent entities, the following comment may be appropriate:

```
/**
 * Compares {@code this} and {@code o} for equality.
 * <p>
 * Any mandatory property may be used in the comparison. If a mandatory
 * property of {@code this} that is used in the comparison is {@code null}
 * an {@code IllegalStateException} will be thrown.
 * @param o reference object with which to compare {@code this}
 * @return {@code true} if {@code this} and {@code o} are equal;
 * {@code false} otherwise
 * @throws IllegalStateException if a mandatory property of {@code this}
 * that is used in the comparison is {@code null}
 */
```

An `IllegalStateException` is thrown to indicate that not enough information is available to make a comparison if any mandatory properties of `this` (none-null) are not set when the method is invoked.

For the `hashCode()` method of persistent entities, the following comment may be appropriate:

```
/**
 * Returns a hash code for {@code this}.
 * <p>
 * Any mandatory property of {@code this} may be used in the hash code. If
 * a mandatory property that is used in the hash code is {@code null} an
 * {@code IllegalStateException} will be thrown.
 * @return hash code
 * @throws IllegalStateException if a mandatory property of {@code this}
 * that is used in the hash code is {@code null}
 */
```

As in the `equals(Object)` comment above, an `IllegalStateException` is thrown to indicate that not enough information is available to derive a distinct hash code from the entity.

- Annotate `equals(Object)`, `hashCode()` and all overridden methods with `@Override`.
- For persistent entities, properties should be read through their getter accessor method in all methods other than the accessor method itself (there should be no need to read the property in the constructor, but if there was, directly accessing the property via its field would be permitted also in such a case).
- Auto generated surrogate primary key properties should not be used in `equals(Object)` or `hashCode()` methods. Instead, the natural "business key" of the object should be determined and used. (From a business perspective, the legality of an object's state is independent of the value of its primary key.)

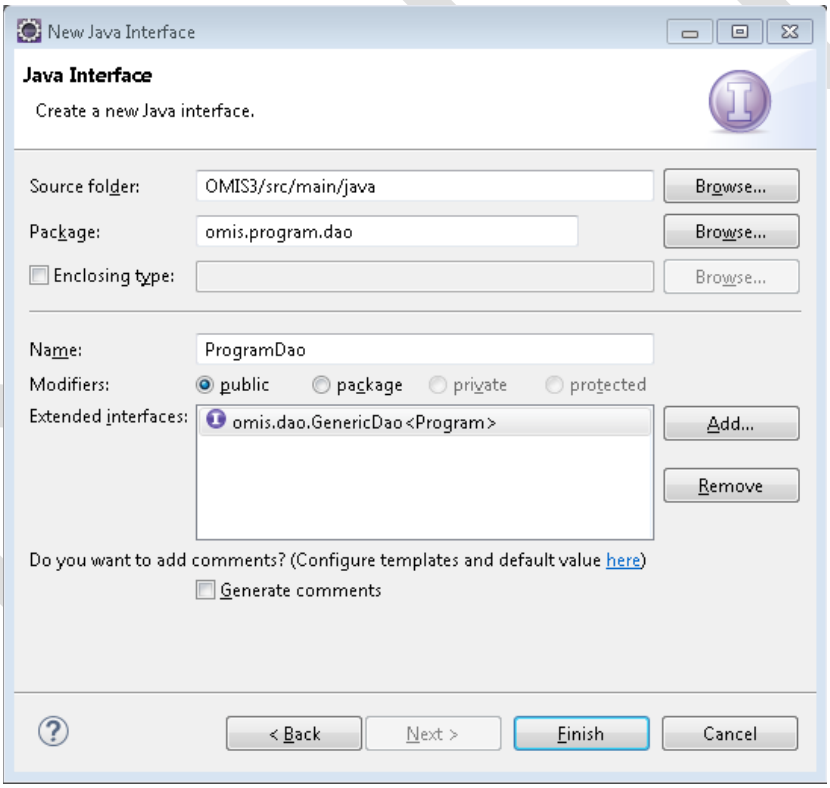
Appendix F: Extending standard implementation of Hibernate DAOs

All common DAO operations such as entity lookup by ID, entity persisting and removing and entity listing can be implemented by extending a standard DAO interface and Hibernate implementation. The standard Hibernate implementation allows an Hibernate session factory to be injected and an entity name to be specified. The standard procedure for extending the Hibernate DAO implementation is as follows:

Create the DAO interface by extending GenericDao<T>

The interface will be used as the reference type in services which use the DAO. The GenericDao interface is generified. It accepts two parameters: <T> the type of the entity. The entity is assumed to have a surrogate primary key property of type `Long`. This property should be named `id`. The GenericDao should be extended by DAO interfaces. It declares all basic read and modify methods.

The DAO for a `Program` entity would be created with the following field values in Eclipse's "New Java Interface" tool dialog box:



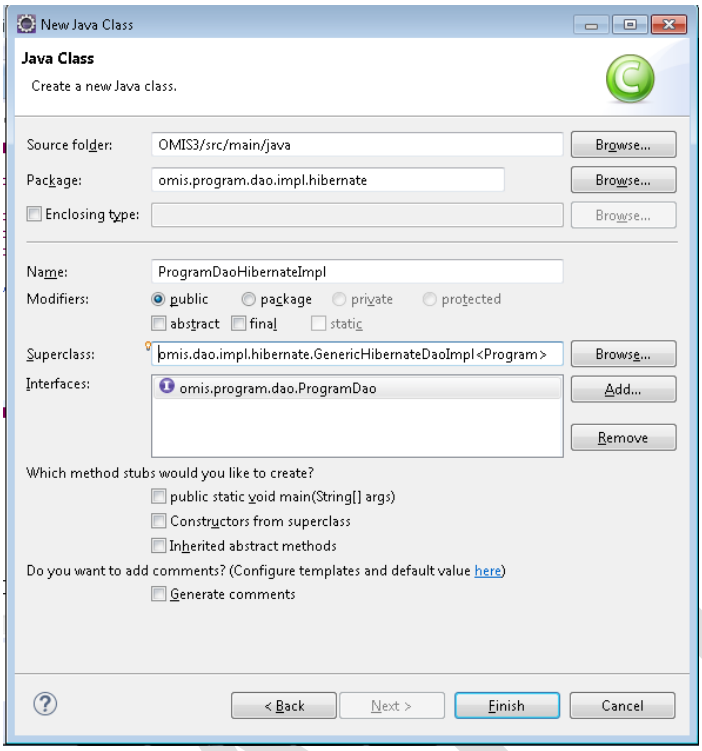
The generated code would look something like the following with look up methods added:

```java
package omis.program.dao;

import java.util.List;

import omis.dao.GenericDao;
import omis.location.domain.Location;
import omis.program.domain.Program;

/**
 * Data access object for programs.
 *
 * @author Stephen Abson
 * @version 0.0.1 (Dec 8, 2015)
 * @since OMIS 3.0
 */
public interface ProgramDao
        extends GenericDao<Program> {

    /**
     * Returns programs provided by location.
     *
     * @param location location
     * @return programs provided by location
     */
    List<Program> findByLocation(Location location);
}
```

Implement the interface and extend the generic Hibernate DAO (in one implementation class)

A standard Hibernate implementation of the generic DAO can be used to provide the basic read and modify methods of the generic DAO interface. Hibernate implementations of DAO classes should implement the **specific** DAO interface of the entity type and extend the standard Hibernate DAO implementation. The standard Hibernate DAO implementation is generified. It accepts the same parameters as the generic DAO interface. The standard Hibernate DAO implementation also extends the generic DAO interface.

The implementation of the Role example above might be created using the following field values in Eclipse's "New Java Class" tool dialog box:



- Note that "Inherited abstract methods" is **unchecked**. To implement the standard DAO, not new methods will have to be defined. The standard implementation will take care of this. ***If "Inherited abstract methods" is left checked, the auto generated sub methods will have to be deleted once the entity object type has been imported into the object's class file.***

The generated code would look similar to the following – method implementations, query names and parameter names are omitted:

```java
package omis.program.dao.impl.hibernate;

import java.util.List;

/**
 * Hibernate implementation of data access object for programs.
 *
 * @author Stephen Abson
 * @version 0.0.1 (Dec 8, 2015)
 * @since OMIS 3.0
 */
public class ProgramDaoHibernateImpl
        extends GenericHibernateDaoImpl<Program> implements ProgramDao {

    /**
     * Instantiates Hibernate implementation of data access object for
     * programs.
     *
     * @param sessionFactory session factory
     * @param entityName entity name
     */
    public ProgramDaoHibernateImpl(
            final SessionFactory sessionFactory, final String entityName) {
        super(sessionFactory, entityName);
    }

    /** {@inheritDoc} */
    public List<Program> findByLocation(final Location location) {
}
```

Allowing Spring to manage the DAO

To have Spring manage the DAO, add a Spring managed bean to the project's application context (applicationContext.xml). The Spring managed Hibernate session factory should be inserted into the Spring managed DAO bean. The application context entry for the role DAO might look similar to the following:

```xml
<bean id="programDao"
      class="omis.program.dao.impl.hibernate.ProgramDaoHibernateImpl">
  <constructor-arg name="entityName" value="omis.program.domain.Program"/>
  <constructor-arg name="sessionFactory" ref="sessionFactory"/>
</bean>
```

When services use DAOs injected by Spring, the DAO will have a standard Hibernate implementation of basic persistence operations. Further, entity specific operations can be declared in the DAO interface (`ProgramDao`) and defined in the Hibernate implementation (`ProgramDaoHibernateImpl`). The implementation should get the Spring managed session factory by calling `getSessionFactory()` and with the returned session factory get the current session by calling `getCurrentSession()`. The session returned by the latter method is the Spring managed Hibernate session. The Spring open session in view interceptor can be used to ensure that the transaction used by the current session is propagated over the entirety of web requests.

- For an OMIS module, have one package for all DAO interfaces. Name it:
  - `<module-package-name>.dao`
- Place DAO interfaces in this package. Name the DAOs:
  - `<entity-name>Dao`
- Place all Hibernate implementations of DAO interfaces in a single package. Name it:
  - `<module-package-name>.dao.impl.hibernate`

- Place the Hibernate implementations of the DAOs in this package. Name them:
    - `<dao-name>HibernateImpl`

Appendix G - Pure Interface Domain Model and Instance Factory

Objectives

A pure interface domain model has the advantage of separating domain model APIs from their implementation. This allows for multiple implementations of business objects providing each implementation strictly honors their API. Thus, the domain model API consists purely of a contract with no implementation code. The implementation used for each business object type is then configured application wide rather than programmatically hard coded. An instance factory is used application wide to instantiate instances of the implementation configured to be used.

Domain Model

API Design and Usage

All of the domain objects of a module should be placed in the domain sub package of the module or sub module. In a purely contractual domain model, interfaces must exclusively be used and therefore only be present in this package. The interface exposes a contract and not implementation details.

The interface will serve as the reference type of the business object throughout each layer of the middle tier as well as Java components of the client/web tier. The components of the middle and client/web tier will work with the contract of the business domain object reference type and not one if its implementations.

- When converting existing modules to use a purely contractual domain model, existing usage of the type being converted should not require modification except where the concrete type business object is instantiated. As interfaces cannot be instantiated, a producer is used to return an instantiated implementation of a purely contractual type (see below).
- The name of the interface must be the name of the type with no prefix or postfix information. This is the name of the reference type of the domain object.
- The API exposed by the purely contractual domain object must work only with data types such as other domain object reference types (in the `domain` package of a module), application data types (in the `datatype` package of the application or a module) or standard Java types (wrapper types must be used over primitive types for property types, "specialist" types such as `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`, `Blob` must always be avoided).
- The methods `equals(Object)`, `hashCode()` and `toString()` should be declared explicitly and documented in the domain model contract (not the implementation), *viz.*, the implementation of these methods must honor an API rather than an implementation contract.
- ***Do not use any data types or exceptions in the package*** `java.sql` ***in a domain model.***

The source code level documentation for the purely contractual domain model (the Javadoc on the interfaces) constituted the entire documentation of the domain model. Each implementation must honor the documented contract in its ***entirety***.

A partial example of a business object type is shown below, comments are omitted:

```
package omis.caution.domain;
// Imports …
/** … */
public interface OffenderCaution extends OffenderAssociable, Updatable {

	/** Sets … */
	void setId(Long id);

	/** Returns … */
	Long getId();

	/** Sets … */
	void setSource(CautionSource source);

	/** Returns … */
	CautionSource getSource();

	/** … */
	@Override
	boolean equals(Object object);

	/** … */
	@Override
	int hashCode();
}
```

- The interface extends `OffenderAssociable`. This contract guarantees that the business object type will support an `offender` property of the type `Offender`.
- The interface also extends `Updatable`. This contract guarantees that the business object type will support an `updateSignature` of the type `UpdateSignature`.
- Both `OffenderAssociable` and `UpdateSignature` enforce the supporting of non-transient properties. Thus, subtypes of each of these types are required to be serializable (each of the latter types extends `java.io.Serializable`). All classes – abstract of concrete – must therefore define a serial version UID (see implementation below).
- The domain object reference type (and not its implementation) is used as the type for composited properties. In this case `CautionSource` (an interface and not an implementation) is used as the reference type for the source property.
- Override the `equals(Object)` and `hashCode()` methods in the domain object type so that the contract and documentation is exposed through the API.
    - A standard comment for `equals(Object)` and `hashCode()` for domain object types is provided in Appendix E.
- A `toString()` contract and documentation should also be included in the domain object reference type API.

Implementation

Implementations of purely contractual domain objects must use only standard Java with no framework dependent code. The implementations must be placed in the `domain.impl` sub package of a module.

- Implementations will differ in the way they perform ***business logic*** only.

Additionally, implementations must only perform business tier functionality.

- Implementations must never directly access tiers of the application other than the business tier. Thus, implementations must never access producers, services, data access objects, web forms, etc.
- As with the API contract, the types used by the implementation should be data types only. This consists of domain object types (from the domain package of a module; not their implementations), application types (from the datatype sub package of the application or a module) or standard Java data types (wrapper types must be used over primitive types for property types, "specialist" types such as `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`, `Blob` must always be avoided).
- ***Do not use any data types or exceptions in the package*** **`java.sql`** ***in a domain model.***

The implementation must never be used explicitly (hard coded) in any layer of the middle tier. Instead, the interface should be used as the reference type. New instances of the implementation types should be retrieved by configuring a producer (see below).

- The `equals(Object)` implementation of the domain object should use the reference type interface and not the implementation type. This is to allow multiple implementations of the same domain object to be used throughout the application, and to be able to be compared accurately. This is especially important in an environment where a managed proxy of an implementation is returned by, for instance, an AOP or persistence framework. See the section below for a further explanation and example.

An example of an implementation of a domain object is shown below. Actual implementation code and full comments are omitted:

```
package omis.caution.domain.impl;
// Imports …
/** Implementation of … */
public class OffenderCautionImpl implements OffenderCaution {
	private static final long serialVersionUID = 1L;
	private Long id;
	private CautionSource source;
	/** … */
	public OffenderCautionImpl() { … }
	/** {@inheritDoc} */
	@Override
	public void setId(Long id) { … }
	/** {@inheritDoc} */
	@Override
	public Long getId() { … }
	/** {@inheritDoc} */
	@Override
	public void setSource(CautionSource source) { … }
	/** {@inheritDoc} */
	@Override
	public CautionSource getSource() { … }
	// Equals and hash code methods …
```

```
        // toString method
}
```

- As `OffenderCaution` is `serializable`, a `serialVersionUID` private field must be defined.
- The internal representation of composited domain objects must be the reference and not implementation type.
- A default constructor is explicitly provided. This may be the only constructor in the implementation.
- Documentation for the method implementations can be inherited (using `@inhertiDoc`).
- Implementations of methods must use the `@Override` annotation for compiler assurance that the correct method is being overridden.

Important note:

- The purpose of this pattern is to separate concerns allowing greater flexibility and configurability in implementing and choosing implementations. ***This pattern does not add additional capabilities or functionality to the domain model.*** For instance, a switch to this model does not allow the domain model to access, for instance the service layer of persistent tier, directly nor does it expose access to the underlying framework, for instance, by allowing injection of framework components.
- Documentation placed in the implementation is implementation specific. It does not pertain to the exposed API of the domain model.
- Documentation of the implementation of a business object API is strongly encouraged.

Implementing `equals(Object)` *and* `hashCode()`, *and* `toString()`

The standard Java methods `equals(Object)` and `hashCode()` must be overridden for every domain object. It is also encouraged that `toString()` also be overridden. These methods form part of the API of the domain model and should therefore be declared and documented as part of the domain model API (in the interfaces) and not their implementation.

- ***Always*** override `equals(Object)` and `hashCode()` in domain objects.
- ***Never*** overload `equals(…)` in domain objects.
- ***Always*** use the accessor of methods of a property in all methods of the implementation of a domain object other than the accessor methods themselves. This includes `equals(Object)`, `hashCode()` and `toString()`.

In the `equals(Object object)` method, `object` must be compared and cast to the reference type of the domain object and not the implementation type. This is to allow for different implementations of domain object types to exist and be compared accurately throughout the application. This is especially important where managed proxy implementations of a domain object might be provided by an AOP container or persistence provider such as Spring AOP or Hibernate respectively.

- Always use the reference type of a domain object (the interface) and never the implementation. This applies even to the implementation of the domain object itself.

Check individual fields in bidirectional associations with other domain objects in the `equals(Object)`, `hashCode()` and `toString()` methods. This is to help prevent two bidirectionally associated domain objects from recursively calling each other's `equals(Object)`, `hashCode()` and `toString()` methods.

- Do not call the `equals(Object)` method of a bidirectionally associated domain object in the implementation of the `equals(Object)` method of a domain object. It may help to assume, defensively, that the `equals(Object)` method is calling the `equals(Object)` method of the current object.
- Equally, avoid using the `hashCode()` and `toString()` method of bidirectionally associated domain objects.
- Audit signatures might be used in `equals(Object)` comparisons and therefore hash code generation in `hashCode()` – but audit signatures are usually insufficient alone .

An example `equals(Object)` and `hashCode()` implementation in `OffenderCautionImpl` is shown below. Note that the association between `OffenderCaution` and `CautionSource` is unidirectional (there is no source to caution relationship), thus, `getSource().equals(that.getSource())` can safely be called without the possibility of a recursive loop of bidirectional `equals(Object)` calls. The same is true of the `hashCode()` method:

```
/** {@inheritDoc} */
@Override
public boolean equals(final Object obj) {
	if (this == object) {
		return true;
	}
	if (!(object instanceof OffenderCaution)) {
		return false;
	}
	OffenderCaution that = (OffenderCaution)that;
	if (this.getSource() == null) {
		throw new IllegalStateException("Source required");
	}
	if (!this.getSource().equals(that.getSource()) {
		return false;
	}
	// Further comparisons using natural key of OffenderCaution
	return true;
}

/** {@inheritDoc} */
@Override
public int hashCode() {
	if (this.getSource() == null) {
		throw new IllegalStateException("Source required");
	}
	Int hashCode = 14;
	hashCode = hashCode * 29 + this.getSource().hashCode();
	// Add hash code of further properties that form natural
	// key of OffenderCaution
	return hashCode;
```

```
}
```

- The reference type of the properties of composited domain object types are used for the comparison and cast in `equals(Object)` (not the implementation). This is to allow for the implementation to be a proxy of the implementation or for the implementation to be switched in different instances of the application.
- Documentation can and should be inherited (using `@inheritDoc`). Additional, implementation specific documentation can be added to the implementation.

Entity Mapping in Hibernate

An entity based configuration is preferred when using a loosely coupled domain model API and implementation. An entity based configuration allows more flexibility in the association between API interface and implementation. The following attributes for mapped domain object are required when such a configuration is used. These are set as attributes of the `<class/>` or `<subclass/>` tags.

Entity Name (`entity-name` Attribute)

This is used as an identifier for a persistent entity matching interface to implementation and persistent representation (part of a table, an entire table or multiple tables). The standard for this identifier is to use the canonical name of the interface for the domain object.

- This is used to identify persistent entities in the persistent tiers. It is also used to identify the type of composited domain objects.

Proxy (`proxy` Attribute)

This is the canonical name of the domain object reference type (interface). The entity name and proxy attribute should hold the same value.

Name (`name` Attribute)

This is the implementation type. This can be altered to switch implementations application wide.

- ***The entity name is used to identify the type of composited domain objects (indicated in the `entity-name` attribute of, for instance, the `<many-to-one/>` association mapping tag).***

A partial example is given below. Note that table information is omitted. The example assumes that `CautionSource`, `UserAccount` and `Offender` are mapped using entity configuration (which might not necessarily be true):

```
<class entity-name="omis.caution.domain.OffenderCaution"
       proxy="omis.caution.domain.OffenderCaution"
       name="omis.caution.domain.impl.OffenderCautionImpl">
  <id name="id" type="java.lang.Long"> … </id>
  <many-to-one name="offender" not-null="true"
               entity-name="omis.offender.domain.Offender"/>
  <many-to-one name="source" not-null="true"
               entity-name="omis.caution.domain.CautionSource"/>
  <component name="updateSignature">
    <many-to-one name="userAccount" not-null="true"/>
    <property name="date" not-null="true"/>
```

```
  </component>
</class>
```

In the mapping same file, a named query may be declared and defined:

```
<query name="findOffenderCautionsByOffender"><![CDATA[
  from omis.caution.domain.OffenderCaution as caution
  where caution.offender = :offender
]]></query>
```

- The entity name is used to identify the type of business object instances to be returned.

Entity Based Configurable Data Access Objects

In order to use entity based configuration, a data access object implementation that reads the entity configuration to identify instances of domain objects in the persistent data store must be used. This is accomplished by the data access object implementation extending GenericHibernateEntityDaoImpl rather than GenericHibernateDaoImpl.

- To use entity based configuration, data access object implementations should extend GenericHibernateEntityDaoImpl.

The GenericHibernateEntityDaoImpl data access object implementation requires the configuration of an entityName property. The value of the entity name property must be the name of the entity. The standard for this name is the canonical name of the domain object reference type. For instance, the entity configurable implementation of a data access object for caution sources would be configured in a manner similar to the following:

```
  <bean id="offenderCautionDao"
class="omis.caution.dao.impl.hibernate.OffenderCautionDaoHibernateEntityImpl">
    <property name="entityName" value="omis.caution.domain.OffenderCaution"/>
    <property name="sessionFactory" ref="sessionFactory"/>
  </bean>
```

- For more information in generic data access object implementations, see Appendix F.

Instance Factory

The purpose of an instance factory is to instantiate instances of a domain object. The advantage to using an instance factory over the Java key word `new` is that the hard coding of implementation types into, for instance, services and controllers, can be avoided. This allows the implementation type of a domain object to be configured and altered application wide by switching the implementation of the instance producer that the application is configured to use or by directly specifying the implementation instantiated by the instance factory.

For instance:

```
OffenderCaution caution = new OffenderCaution(); // Without producer
```

Is replaced by:

```
OffenderCaution caution = this.offenderCautionInstanceFactory
        .createInstance();
```

The implementation type returned by the `createInstance()` method of `offenderCautionInstanceFactory` can be configured.

This pattern is implemented using the parameterized instance factory interface and a configured instance of a generic implementation.

- Instance factories must be used only to instantiate domain objects and not, for instance, other Java objects used in the application such as web forms.

API Design and Usage

An instance factory for offender cautions would be declared as follows:

```
InstanceFactory<OffenderCaution> offenderCautionInstanceFactory;
```

Such an instance factory would be used in a service or controller as follows:

```
OffenderCaution caution = offenderCautionInstanceFactory.createInstance();
```

Hibernate Implementation

A single Hibernate implementation is used for instance factories for all entities. The implementation is:

```
omis.instance.factory.hibernate.impl.InstanceFactoryHibernateImpl
```

An instance of this implementation must be configured per entity.

Spring Dependency Injection Configuration

Instance factory implementation instances must be configured per entity as managed beans in the Spring application context configuration for the module. The instance factory can then be injected into components such as services and controls that may require an instance of the produced business object implementation. Using XML, a Hibernate entity configurable offender caution instance factory would be configured as follows:

```
<bean id="offenderCautionInstanceFactory"
class=" omis.instance.factory.hibernate.impl.InstanceFactoryHibernateImpl

">
  <constructor-arg name="instanceFactoryDelegate"
ref="instanceFactoryDelegate"/>

  <constructor-arg name="entityName"
value="omis.caution.domain.OffenderCaution"/>
</bean>
```

- The `instanceFactoryDelegate` is used to instantiate the entity implementation . This property should always be set to the `instanceFactoryDelegate` reference.

- The `entityName` property should match the value of the entity-name attribute of the class or subclass as configured in the HBM (see above).

Appendix H - Web Tier Folder Structure and Request Mapping Standards

Controllers

Packages containing web controllers should follow the following naming convention:

```
omis.<module-name>.web.controller
```

Main controller should take the name of the module and not the perceived main component of the module, other controllers should be component specific and should take the name of the subcomponent:

`OffenderController` – Offender related operations, main controller

`EmploymentController` – Employment related operations, main controller – purpose of employment module is to track employment (not, for instance, employees – that is a requirement additional to the tracking of employment).

`HealthCareRequestController` – controller for subcomponent of medical module

Forms

Forms to capture information from the web tier should be placed in a package with the following naming convention:

```
omis.<module-name>.web.form
```

For instance:

```
omis.caution.web.form.OffenderCautionForm
```

```
omis.medical.web.form.HealthCareRequestForm
```

The names used should mirror business objects in the `domain` sub package of the module.

- Forms should only contain properties editable on the view form used to enter the form details. A form should not contain the database ID of the business object for which the form is designed to edit. Nor should the form contain other, none visible properties such as the offender or update and creation signatures.
    - An appendix on this would be good…

URL Mappings

- Main controller should take module name – should form parent URL path:
  /employment
  /medical

- Finer grained components in more complex modules should have their own controller with their own URL sub path appended to the parent path:
  /medical/healthCareRequest.html
- Methods should be mapped to URL actions, HTTP method (POST, GET) vary with the same URL path:
  - /create.html, method = GET – display form to create new record
  - /create.html, method = POST – save newly entered record
  - /edit.html, method = GET – display form to edit existing record
  - /edit.html, method = POST – save changes to existing record
  - /remove.html, any method – remove record
  - /list.html, any method – display list of all records

To create a new employment record, invoke:

/employment/create.html?offender=<offenderNumber>

To edit an existing employer invoke:

/employment/employer/edit.html?employer=<ID>

To remove a health care request:

/medical/healthCareRequest/remove.html?healthCareRequest=<ID>

- ***Path component names should be camelCased!***

View Naming and Locations

All JSPs should be placed in WEB-INF/views or an appropriate subfolder of WEB-INF/views. No JSPs should be directly exposed to the client (all should be accessible only via a dynamic web module such as the request method of a Spring MVC Controller).

Views for modules should be placed in a subfolder of WEB-INF/views named after the module. Example of such folder names are:

- /WEB-INF/views/caution
- /WEB-INF/views/employment
- /WEB-INF/views/offender

All JSP view and therefore view file names should be camelCased. Two file names are for listing screens and details screens, respectively:

- /WEB-INF/views/caution/list.jsp
- /WEB-INF/views/caution/edit.jsp

These two JSPs represent entire HTML documents/dynamic webpages to display, list and edit records of the type which is the focus of the module (OffenderCaution). They therefore reside at the module level

folder of the module view path. Other listable and editable components of the module should have views placed in their own subpath:

- /WEB-INF/views/caution/source/edit.jsp – create or update CautionSource instances
- /WEB-INF/views/caution/description/list.jsp – lists CautionDescription instances

It is encouraged that elements of a view which may be useful in their own right be separated into separate JSP files containing not full HTML web pages but rather snippets of screens that might be useful in multiple JSP view files. These snippets should be places in the "includes" subfolder of a module or module subcomponent folder:

- /WEB-INF/views/caution/includes/editForm.jsp – contains the entire <form/> element to edit cautions
- /WEB-INF/views/caution/source/includes/listTable.jsp – table to display a list of caution sources
- /WEB-INF/views/caution/description/includes/listTableBodyContent.jsp – content of a table body (<tbody/>) element for listing caution descriptions (a series of table rows (<tr/>s) containing table cells (<td/>s))

Snippets may include other snippets so that the listTable.jsp view for caution source would include the listTableBodyContent.jsp for content source listing inside its <tbody/> tag.

The snippet name should include the actual name of the screen element if a complete element or the screen element plus the camel cased postfix of content if the content of the snippet is intended to the content of an element (for instance, listTableBodyContent.jsp in the example given above).

- ***View names and view file names- including snippets - should be camelCased!***

Web Resource Naming and Location

- Absolute paths should always be used to include resources in a JSP. The `contextPath` property of the Servlet `request` from the `pageContext` should be used to construct the absolute path:

```
<link rel="stylesheet" type="text/css"
href="${pageContext.request.contextPath}/resources/common/style/general.css"/>
```

- Static style sheets and functional scripts should be named using camelCase, for instance:
  - /resources/medical/scripts/healthRequestRoutingCategoryLoader.js
  - /resources/caution/style/caution.css – for OffenderCaution editor styling
  - /resources/caution/style/sources.css – for styling CautionSource list
- For static script files containing only first class components, use the component name:
  - /resources/common/scripts/EventRunner.js
- Dynamic scripts should be exposed using URLs that follow the naming conventions of static scripts. The dynamic views used to output the script, however, should be placed in:
  - /WEB-INF/views/scripts/js - for dynamic (server side generated) JavaScript
  - /WEB-INF/views/style – for dynamic (server side generated) CSS

  The JSP view should have the same name as exposed through the dynamic web module:

- Invoking the URL for the dynamic script /resources/common/scripts/SessionConfig.js output the dynamically generated script content in /WEB-INF/views/common/scripts/js/SessionConfig.jsp. Note that the exposed URL takes the extension of .js, the actual view the extension of .jsp.

- ***Web component naming and folder structure does not necessarily mirror middle tier class names and package structuring!***

Appendix I - Web Form Beans in Spring MVC

Purpose

Web form beans capture information to populate persistent entities and are used to display information retrieved from the persistent tier in entity instances.

Web forms must:

1. Contain only information editable by the user. This does not include:
    a. Persisted entity instance IDs (surrogate primary key)
    b. "Hidden" data such as offenders or creation signature information.
2. Contain associated entity properties, single or collection types, and not their IDs.

Used with Spring MVC, web forms allow the exchange of redundant, duplicate and potentially conflicting information to be avoided. They also allow information to be manipulated using an object oriented model.

Implementation

Creating the Form Bean

Create a JavaBean type with a name that best describes the information to be edited and add the postfix "Form". In many cases, this will correspond to the name of the persistent entity. For instance, a form to create and edit health care requests should be called:

```
HealthCareRequestForm
```

The form must be places in a package with a name following this convention:

```
omis.<module-name>.web.form
```

The health care request form would therefore be placed in:

```
omis.health.web.form
```

Assuming health is the module package name of the health module.

The bean must implement serializable and therefore declare a serializable ID:

```
public class HealthCareRequestForm implements Serializable {
	private static final long serialVersionUID = 1L;
}
```

Adding Fields to the Form Bean

Properties reflecting information directly editable by the user and displayed on the client tier form can be added to the form bean.

- ***<u>The form bean must not include fields which are not directly editable by the end user. Such fields include:</u>***

- ***Persistent entity ID/auto generated surrogate primary key***
- ***Offender***
- ***Creation or update signature***

The form bean should include properties of the type of information editable by the user. For instance, a health care request form might have a `category` property of the the `HealthRequestCategory` type:

```
private HealthRequestCategory category;
public void setCategory(HealthRequestCategory category) {
	this.category = category;
}
public HealthRequestCategory getCategory() {
	return category;
}
```

Note that this property is of the actual instance type and not a primary key reference.

- ***Do not put primary key references on web forms.***

If the client form has a multiple choice field, a collections property of the type `Set<T>` can be used as a form bean property. For instance, the health care request form might have a `routingCategories` property of the type `Set<HealthRequestRoutingCategory>`:

```
private Set<HealthRequestRoutingCategory> routingCategories = …;
public void setRoutingCategories(
	Set<HealthRequestRoutingCategory> routingCategories) { … }
public Set<HealthRequestRoutingCategory> getRoutingCategories() { … }
```

Properties of none composite types should use wrapper over primitive types (Integer over int, Long over long, etc).

Controller configuration

A basic web form bean will most commonly be used in four requests. For an explanation of the function and purpose of these requests, see Appendix <UNKNOWN>:

Create (GET method) – displays a form to capture new information

Create (POST method) – accepts submitted form of captured new information to be saved

Edit (GET method) – displays a form to modify existing information

Edit (POST method) – accepts submitted form of modified existing information to be updated

For create requests, information not editable by the user via the client tier form and not determined on submission must be passed as request parameters. This information is most commonly the offender:

/heatlh/healthCareRequest/create.html?offender=99999

No field will exist on the web form to edit the offender and this value will not be determined when the form is submitted. Instead, it is more likely that this information will be determined from another page – a listing or search screen – and be passed as a request parameter via, for instance, and HTML link.

For edit request, the only additional information that is required other than that editable on the client form is the record to be edited itself. This should also be passed as a request parameter:

/health/healthCareRequest/edit.html?request=32

- Note that in each case (offender and health care request alike), a property editor resolves ID to entity instance. Thus, entities are passed and not IDs. The controller method for the create request will accept an `Offender` object (not an offender number `Integer`) and the edit method will accept a `HealthCareRequest` object (not an ID `Long`).
- When a Spring MVC form is submitted using the default action attribute value (by not setting it) the form will be submitted to the URL from which it was originally invoke with the full query – including GET parameters. Thus, the offender and request objects above will be available in both GET and POST/display and submit requests. It is therefore crucial that this information not be included as form bean properties.

The method signatures for create and edit controller request for health care requests – editable via the health request form bean - would therefore be the following:

```
@RequestMapping(value = “/create.html”, method = RequestMethod.GET)
public ModelAndView create(
	@RequestParam (value = “offender”, required = true)
		Offender offender);
```

This method would create a new, blank health care request form and display it to the user (by adding it to the returned model).

```
@RequestMapping(value = “/create.html”, method = RequestMethod.POST)
public ModelAndView save(
	@RequestParam(value = “offender”, required = true)
		Offender offender,
	HealthCareRequestForm requestForm);
```

This method would copy the values of the request form and the none-editable offender property to a new instance of a health care request and persist it. The method would probably also set other none-user-editable properties such as the update and creation signature.

```
@RequestMapping(value = “/edit.html”, method = RequestMethod.GET)
public ModelAndView edit(
```

```
        @RequestParam (value = “request”, required = true)
              HealthCareRequest request);
```

This method would copy the editable values of the request to a new request form for editing by the user (by adding it to the returned model).

```
@RequestMapping(value = “/edit.html”, method = RequestMethod.POST)
public ModelAndView update(
      @RequestParam(value = “request”, required = true)
            HealthCareRequest request,
      HealthCareRequestForm requestForm);
```

This method would copy the values of the request form to the request object and persist the updates. This method may also update additional, none-editable fields such as the update signature.

Note that in each GET request the same web form can be used in without modification:

```
<form:form commandName=”requestForm”>
      <!-- Fields, etc  -->
</form:form>
```

- Model attributes such as collections for drop downs must be added in the same way in which they would were persistent entities to be used as the command object.

Appendix J - Web Tier Element Naming and Localization Keying

All element IDs, classes, message content keys, JSPs, JSP snippets, and JSP directories and subdirectories should be in camelCase (as should URLs).

Element	Design Time Markup	Runtime Markup if different	ID	Class if different	Message Content Key	JSP Include Snippet**	Remarks
Title	<title/>		Not required, target <title/>		+ Title		Title and main header content should be the same
Main Header	<h1/>		Not required, target <h1/>		+ Title		Title and main header content should be the same
Other headers	<h2/>, <h3/>, etc		+ Header		+ Header		
Spring Form	<form:form/>	<form/>	Will take ID of command object name	Command object name	N/A	+ Form eg, editForm	Always name command object
Spring Label	<form:label/>	<label/>	path + Label		path + Label		Value of “for” attribute should be ID of target element
Spring Input	<form:input/>	<input type=”text”/>	Path will be used		N/A		
Spring Select	<form:select/>	<select/>	Path will be used		N/A		Target for content update

Spring Options	<form:options/>	<option/>	Nothing? Check!		N/A		For snippets, code HTML manually
Spring Option	<form:option/>	<option/>	Nothing? Check!		If required, textual representation of value + Label	+ Options	For snippets, code HTML manually
Spring Checkboxes	<form:checkboxes/>	☐	UNKNOWN what is produced, Use path + index, eg, request1 manually		N/A		For snippets, code HTML manually
Spring Checkbox	<form:checkbox/>	☐	path + index, eg, request1		If required, textual representation of value + Label	+ Checkboxes	Name will be path For snippets, code HTML manually
Spring Radio Buttons	<form:radiobuttons/>	<input type="radio" />	path + index will be used		N/A		For snippets, code HTML manually
Spring Radio Button	<form:radiobutton />	<input type="radio" />	path + index, eg, request1		If required, textual representation of value + Label	+ RadioButtons	Name will be path For snippets, code HTML manually
Spring Errors	<form:errors />	Usually	+ Errors	+ Errors and	path + Error		

				errors for all			
Table	<table/>		+ Table		N/A	+ Table eg, listTable	Every table should have a <tbody/> element
Table Head	<thead/>		+ TableHead		N/A		
Table Row	<tr/>		Depends on content		N/A	+ TableRow	
Table Header	<th/>		+ Header		+ Header	+ TableHeader	
Table Body	<tbody/>		Pluralized content – eg, requests, cautions		N/A	+ TableBody eg, listTableBody	Target for content update
Table Cell	<td/>		Depends on content		If required, textual representation of content + Label	+ TableCell	
List	 or 		Pluralizaed content – eg, links; or name – eg, toolbar		N/A	+ List	
List Item			Depends on content		If required, textual representation of content + Label	+ ListItem	
Link	<a/>		Depends on content		If require	+ Link	

					d, textual represe ntation of content + Link		
Paragrap h	<p/>		Content – pluralized accordingly , eg, categories.		Content will vary – text should be contain ed in child elemen ts		Target for content update
Input buttons	<input type=”submi t”/> <input type=”reset” />		+ Button		+ Label		Target for enable/ disable

*Mandatory

**The snippet may contain the content of an element but not the element itself. In which case the name of the snippet should have the postfix “Content” – `listTableBodyContent.jsp` is an example

When manually outputting input elements such as radio boxes and check boxes, the index can be retrieved from the `index` property of the `varStatus` variable of the `<c:forEach/>` tag:

```
<c:forEach varStatus=”status”>
      <c:out value=”${status.index}”/>
</c:forEach>
```

Appendix K: Package Names for Spring/Hibernate Implementation

Package Name	Description	Component Contents	Example Content
omis.offender	Module parent package Module submodule parent package	Module subpackages Module sub module parent package No classes	omis.offender.dao omis.offender.domain
omis.offender.dao	Module DAOs	Module DAO interfaces DAO implementation subpackage	OffenderDao omis.offender.dao.impl
omis.offender.dao.impl	Module DAO implementations	Module DAO implementation subpackages No classes	omis.offender.dao.impl.hibernate
omis.offender.dao.impl.hibernate	Hibernate implementation of module DAOs	Hibernate DAO implementation classes	OffenderDaoHibernateImpl
omis.offender.datatype	Module data types	Module data type classes Module data type user type subpackage	<<CLASS EXAMPLE RQUIRED>> omis.offender.datatype.usertype
omis.offender.datatype.usertype	Hibernate user types for module data types	Hibernate user type classes	Deprecated – user types should be avoided
omis.offender.domain	Module domain model	Module domain model Domain model helper class subpackages Interface only	Offender
omis.offender.domain.impl.delegate	Module domain model delegates	Module domain model delegate helper classes	OffenderPropertiesDelegate
omis.offender.domain.impl	Module domain model implementations	Implementation of domain model types	OffenderImpl
omis.offender.exception	Module business exceptions	Business exception classes	
omis.offender.service	Module services	Module service interfaces Module service	OffenderService omis.offender.service.impl

		implementation subpackage	
omis.offender.service.impl	Module services implementations	Module service implementation classes	OffenderServiceImpl
omis.offender.web	Module web component parent package	Module web component subpackages No classes	omis.offender.web.controller omis.offender.web.form
omis.offender.web.controller	Module Spring MVC web controllers	Module Spring MVC web controller classes	OffenderController
omis.offender.web.form	Module web forms	M web forms for use with Spring MVC, Struts, etc	OffenderForm

Appendix L: Report Services

Report services are an efficient way of retrieving result sets of information where associations between entities inadequately represent the way in which the information is to be reported and/or the instantiation of the entire graph of the related entities is not required (and would be wasteful). Four components are required to implement report services:

1. Report objects – type safe, read only concrete Java classes which hold values.
2. Report service API – an interface that exposes methods which return report objects or collections (`Lists`) of report objects. One report service API should be created per module. It should take the name of the module with the postfix `ReportService`.
3. Report service implementation – a framework/implementation specific concrete class which honors the contract of the report service API.
4. Framework/implementation specific data retrieval – this may take the form of an ORM framework specific query.

Report Objects

Report objects hold values of related information specific to a purpose. The purpose should be indicated in the report object name and API documentation. Each report object should be prefixed with a noun that describes its purpose – such as summary or overview. Report objects should be concrete, final and immutable. In order to achieve this, report object should:

- Not be interfaces or abstract.
- Be declared final.
- Not expose client or data tier concerns.

Report object data members should

- Be declared final and private.
- Have only a getter method accessor method.
- Have a getter which returns a deep copy of mutable data types (such as `java.util.Date`).
- Not be entities or components. Rather, the individual properties of the entities and components queried should be used as the data members of the report object.
- Be set once in a single constructor.

Report objects must be serializable and in most perceivable cases do not require an `equals(Object)` or `hashCode()` method.

Report objects must have a single constructor that sets all properties.

Report objects should light way and easily serializable to allow them to be passed to the web tier or a reporting technology (such as iReports).

- Report objects must be placed in the report sub package of a module – for instance, `omis.offender.report`.

Report Service API ("Report Service")

The report service API should expose methods that allow the querying of information. Report service API methods may return either single instances of report objects or collections (as ordered `Lists`). The verb of the method should indicate the nature of the information returned – such as summarize for a summary report object. Entity instances – and not ID values of entity instances – should be passed to report service methods.

A single report service API should exist per module. It should be named after the module with the postfix `ReportService`.

- The report service API should be placed in the `report` sub package of a module along with the report objects – for instance, `omis.offender.report`.
- The report service API should not expose client or data tier concerns.

Hibernate Implementation of Report Service API

The Hibernate implementation of a report service consists of two components – a concrete class which implements the report service API and a HBM file exclusively containing queries invoked by the methods of the report service implementation class.

Implementation Class

Unlike business services, Hibernate implementations of report services can have direct access to the Hibernate `SessionFactory`. A `SessionFactory` must be injectable via a setter method (named `setSessionFactory` taking a single `SessionFactory` parameter named `sessionFactory`).

- Hibernate implementations of report services should be placed in the `report.impl.hibernate` sub package of a module and should be named after the report service with a postfix of `HIbernateImpl`. For instance, the canonical name of the Hibernate implementation of the report service for the offender module would be:

```
omis.offender.report.impl.hibernate.OffenderReportServiceHibernateImpl
```

Query HBM

The Hibernate query HBMs should be placed the `report` sub package of the module folder in the resources source folder. The HBM file should be the name of the module with a postfix of Queries. For instance:

```
omis.offender.report.OffenderQueries.hbm.xml
```

The HBM should contain only named queries (no mapped entities).

Appendix M: Miscellaneous Development Tips

Check the javadoc

- You might be surprise what you find.

Place example usage of an API in your own javadoc

- *This can be priceless*. Use a <code /> tag. Look at the ways in which you use your own APIs, use a simplified version as your example usage. Place multiple examples of different kinds of usage in your javadoc if you think of examples of different kinds of usage.
- Look for instructive parts of javadoc that you have written and supplement them with an example.

Initialize a local variable as close to where it is declared as possible

- Only initialize a local variable with null if there is a chance that it will not be set (and therefore need to be tested for null). If it is expected that a local variable will never be null, do not initialize it with null – the compiler will catch the use of uninitialized variables at compile time. This reduces the chance of null pointer exceptions. Initialize a local variable to null only if, after a specific process, it may still be null and require checking for null.

Reconsider the design of an entity of it contains many properties of simple types

- For instance, lots of Strings, Integers, Longs, BigDecimals or Dates (floats, doubles, Floats and Doubles should be avoided) – *see standard XXX*
- Group the fields into subcomponents
- Fine grained object model

Modularize large processes into smaller, separate and specific [micro processes]

- Entire method should fit on less than one screen unless mundane and repetitive – eg, huge switch/case statement, but even then, a better, polymorphic design might be possible
- Input data types provide minimum functionality for what is needed to be done – use interfaces not concrete implementation types
- Assists in modularization
- Encourages reusability

Design to provide for usage that reads like a "human language"

As much as possibly, the API of a software module should be designed for usage which reads similar to the target, "human" language of the project. For example, it is clear what the following piece of code does without having to read the implementation:

```
// Good code – well worded, clear, well thought out
if (proposedEvent.conflictsWith(savedEvent)) { … }
```

In keeping with this best practice, the `conflictsWith` method should perform one action. It should compare two events to see if they conflict and return the result. It is up to the event itself to decide and document exactly what is meant by two events conflicting. The behavior may even be polymorphic – in

fact such a design strategy is encouraged. This is a secondary concern for the client. As far as the client is concerned conflicts between two events are identified – it doesn't matter how they conflict, only whether or not they do.

A poorly designed API for the same functionality may looks something like the following:

```
// Bad code – poorly designed, inflexible, nonsensical language
if (ConflictDetectionUtil.detectConflict(proposedEvent, savedEvent) { … }
```

This does not read like a "human" language. What is more, this design is inflexible. The definition of "conflict" in this sense has to be known by the `ConflictDetectionUtil` utility class. It cannot be delegated as easily to the objects in question. The previously given "good" example allows each event type to define its own behavior. This has the additional positive effect of contributing to the building of a richer domain model by including business logic in self-contained objects.

Recommendation

- Implement requirements in Java code!

Appendix M: Common Errors

Spring MVC

HTTP Status 400- description: The request sent by the client was syntactically incorrect ().

A required parameter is missing. The parameter that is required is defined in the method corresponding to the URL call. This method is a Spring controller method.