

Transforming models into code (Basics)

© 2009 Huascar A. Sanchez. All rights reserved.

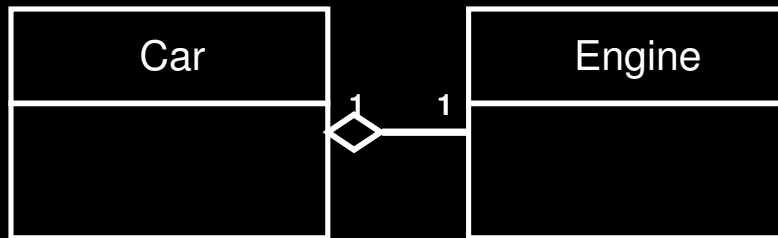
By
Huascar A. Sanchez

Agenda

- Forward Engineering
- Mapping Activities
 - Mapping types to code
 - Mapping classes to code
 - Mapping associations to code
 - Mapping contracts to code
- The end

Forward Engineering

- An activity intended to transform a set of model elements into a set of source code statements. i.e.,



=

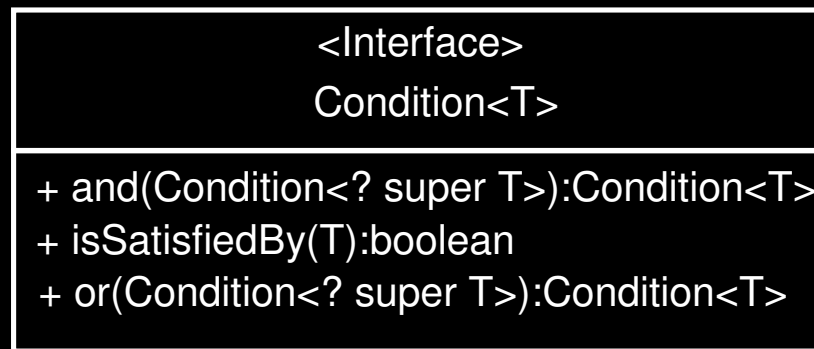
```
public class Car {
    private final Engine engine;
    ...
    @Inject
    public Car(Engine engine) {
        this.engine = engine;
    }
}
```

Forward Engineering

- Forward engineering is applied during the following activities:
 - Mapping types to code
 - Mapping classes to code
 - Mapping associations to code
 - Mapping contracts to exceptions

Mapping types to code

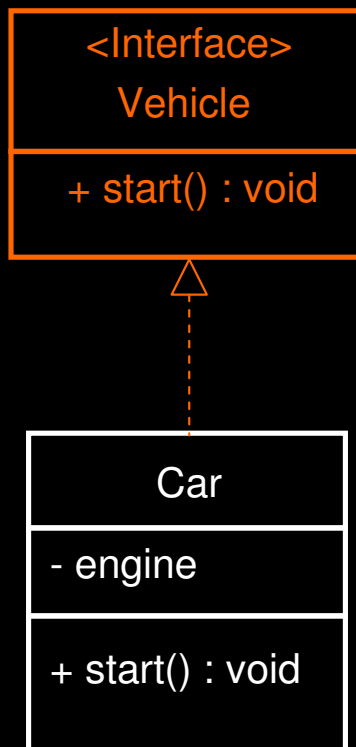
- Def: A type names its public interfaces.



```
public interface Condition<T> {  
    Condition<T> and(Condition<? super T> c);  
    boolean isSatisfiedBy(T thing);  
    Condition<T> or(Condition<? super T> c);  
}
```

Mapping classes to code

- Def: A class is a language construct intended to create objects. It implicitly or explicitly implements a type.



=

```
public class Car implements Vehicle {
    private final Engine engine;
    ...
    @Inject
    public Car(Engine engine) {
        this.engine = engine;
    }

    public void start() {
        engine.start();
    }
}
```

Mapping associations to code

- *Or* association (a.k.a., inheritance)



```
public class Entry {
    private Object value;
    ...
    public record(Object o) {
        value = o;
    }
}
```

```
public class FormattedEntry
    extends Entry {
    private final Format format;
    ...
    public format() {
        value = format.enhance(value);
    }
}
```

Mapping associations to code

- *And* association (a.k.a., aggregation, composition)

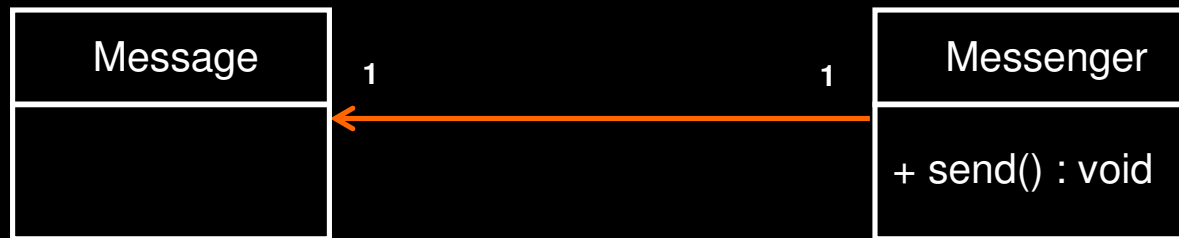


```
// normal aggregation
public class Car {
    private final Engine engine;
    ...
    public Car(Engine engine) {
        this.engine = engine;
    }
}
```

```
// composition
public class Car {
    private final Engine engine;
    ...
    public Car() {
        this.engine = new CarEngine();
    }
}
```

Mapping associations to code

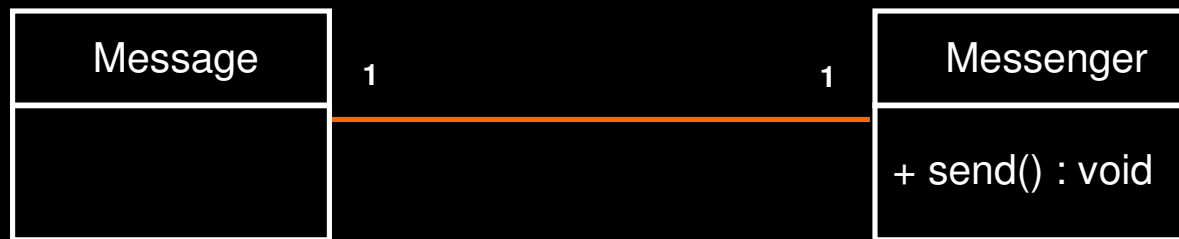
- *Unidirectional*, one-to-one association



```
public class Messenger {
    private final Message m;
    public Messenger() {
        m = new Message();
    }
    ...
    public void send() {
        deliver(m);
    }
}
```

Mapping associations to code

- *Bidirectional*, one-to-one association

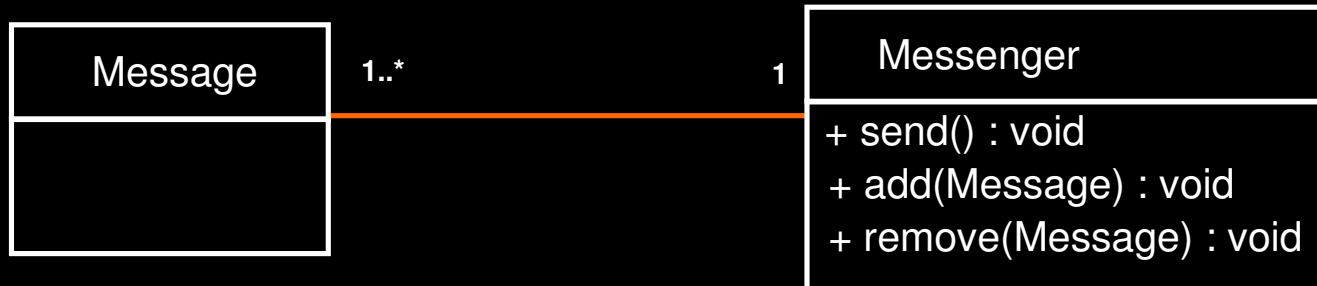


```
public class Message {
    private final Messenger m;
    public Message(Messenger m) {
        this.m = m;
    }
    ...
    public Messenger get() {
        return m;
    }
}
```

```
public class Messenger {
    private final Message m;
    public Messenger() {
        m = new Message();
    }
    ...
    public void send() {
        deliver(m);
    }
}
```

Mapping associations to code

- *Bidirectional*, one-to-many association

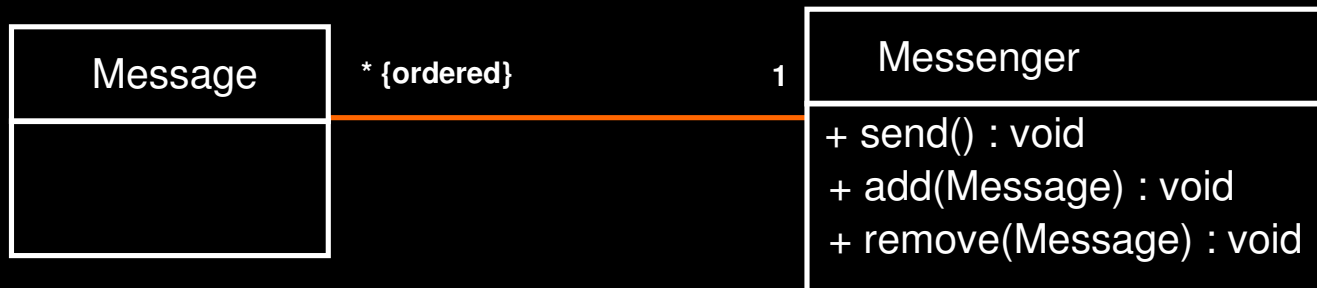


```
public class Message {
    private Messenger m;
    public Message(){
        m = null;
    }
    ...
    public void setOwner(
    Messenger owner
    ) {
        m = owner;
    }
}
```

```
public class Messenger {
    private final Set<Message> m;
    public Messenger(){
        m = new HashSet<Message>();
    }
    public void add(Message each){
        each.setOwner(this);m.add(each);
    }
    public void remove(Message each){
        m.remove(each);each.setOwner(null);
    }
    ...
}
```

Mapping associations to code

- *Bidirectional*, one-to-many (ordered) association

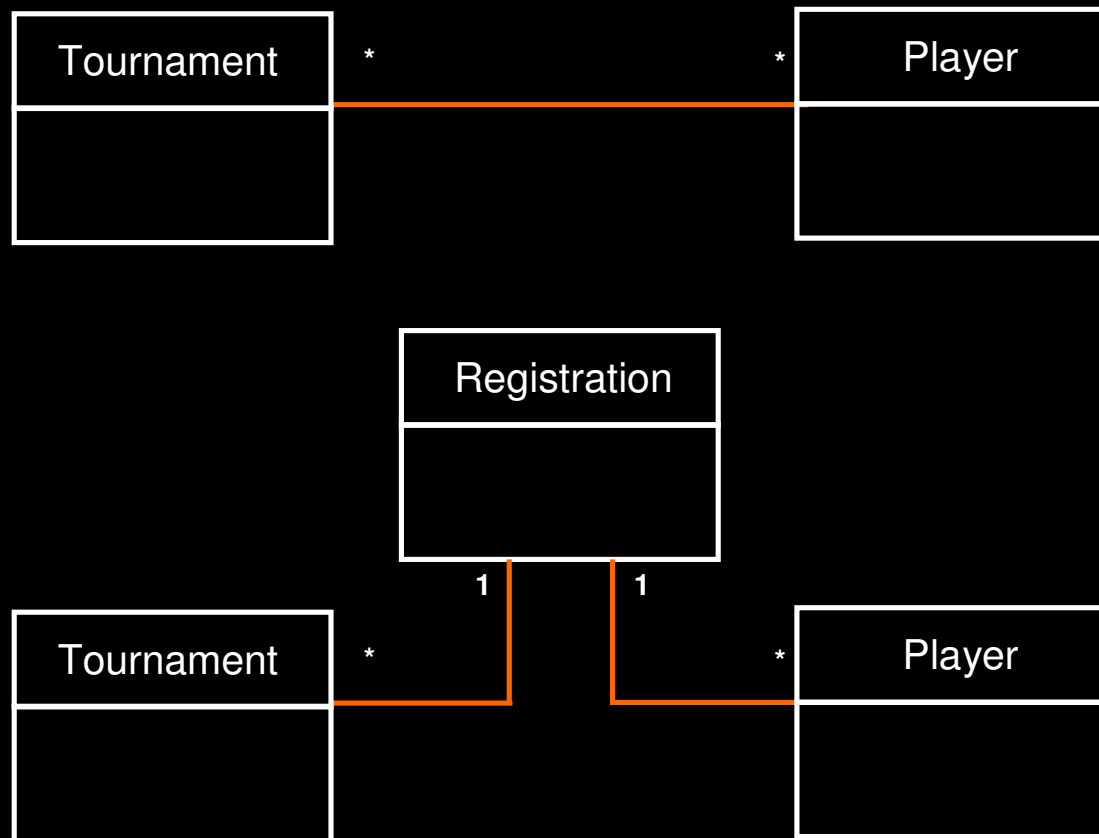


```
public class Message {
    private Messenger m;
    public Message(){
        m = null;
    }
    ...
    public void setOwner(
    Messenger owner
    ) {
        m = owner;
    }
}
```

```
public class Messenger {
    private final List<Message> m;
    public Messenger(){
        m = new ArrayList<Message>();
    }
    public void add(Message each){
        each.setOwner(this);m.add(each);
    }
    public void remove(Message each){
        m.remove(each);each.setOwner(null);
    }
    ...
}
```

Mapping associations to code

- *Bidirectional*, many-to-many association



please see previous slides for implementation details.

Mapping contracts to exceptions

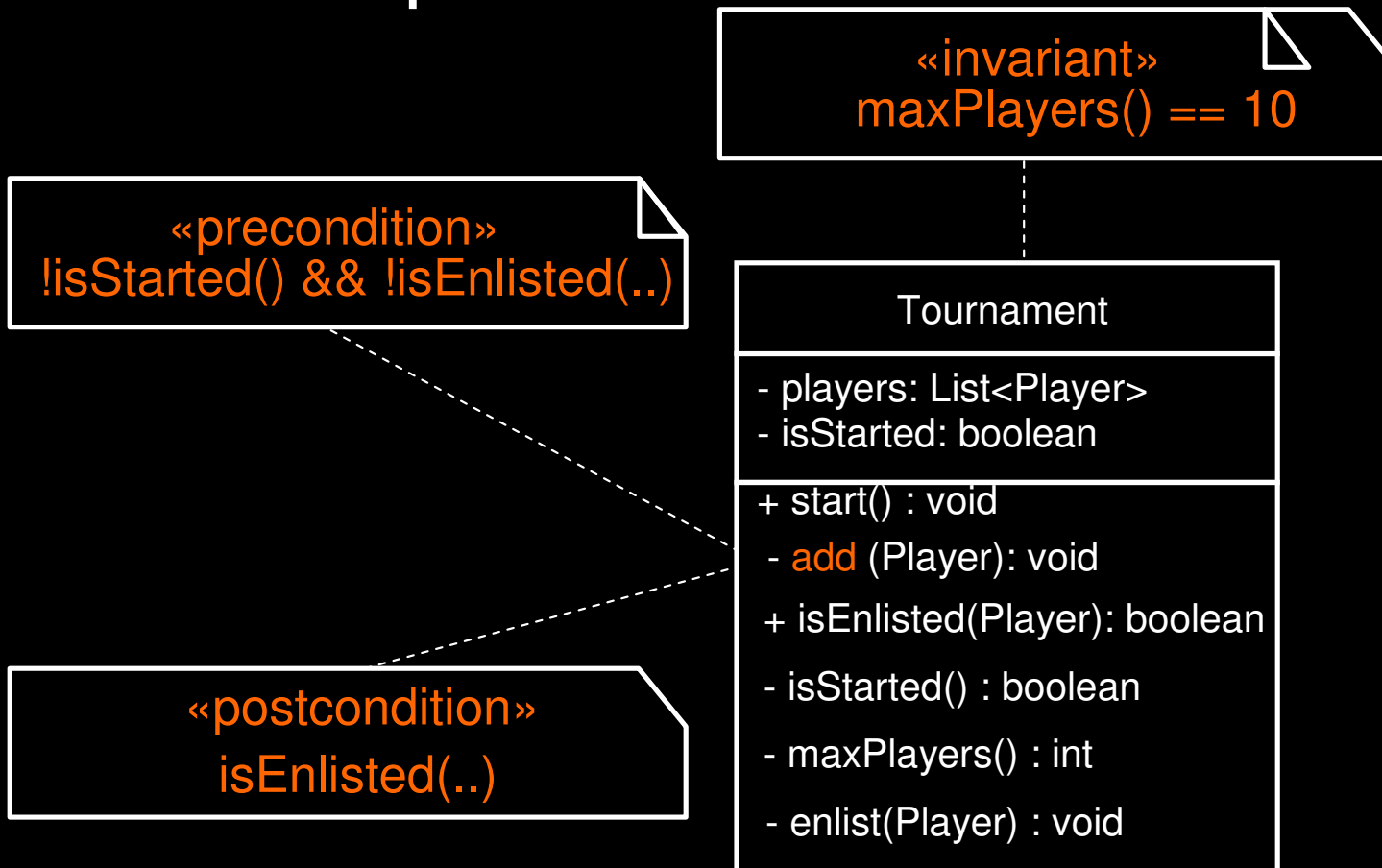
- Throw exceptions when violations occurred.
- Java does not include built-in contract support.
- However, Java exceptions can be used for signaling & handling contracts violation.
- Use the *try-catch* to handle the exceptions.

Mapping contracts to exceptions

- Type of contracts
 - Pre conditions: to be checked before beginning of a method.
 - Post conditions: to be checked at the end of the method.
 - Invariants: same as the post conditions.

Mapping contracts to exceptions

- Example



Mapping contracts to exceptions

```
public class Tournament {
    private final List<Player> players;
    ...
    public void add(Player p)
    throws PreCondException,
    PostCondException,
    InvariantException {
        if(isStarted() && isEnlisted(p)){
            throw new PreCondException(..);
        }
        enlist(p);
        if(!isEnlisted(p)){
            throw new PostCondException(..);
        }
        if(maxPlayers() > 10){
            throw new InvariantException(..);
        }
    }
    ...
}
```

- The End

- Any questions?