Aspectual Collaborations

Modules and Aspects

Johan Ovlinger

February 5, 2002

Modules and Decomposition

- Modules decompose programs into encapsulated units.
- The encapsulation interfaces are strong – cannot be broken (a module could have several such interfaces).
- Architectural decomposition of program
- Promotes reuse and separate development.

Representative examples are: Units.

Concerns and Aspects

- Concerns decompose the program into overlapping units.
- Functional decomposition of program.
- Concerns seldom fit module boundaries.
- Promotes separate specification of overlapping tasks.

Aspectual Collaboration Motivation

Some weaknesses of Aspects without Modules

- Aspects cannot be analyzed in isolation: need global insight into program.
- For this reason, Aspects are hard to reuse in different programs.

Some weaknesses of Modules without Aspects

- Tangling / Scattering
- Puts interaction between concerns into code, rather than into module language

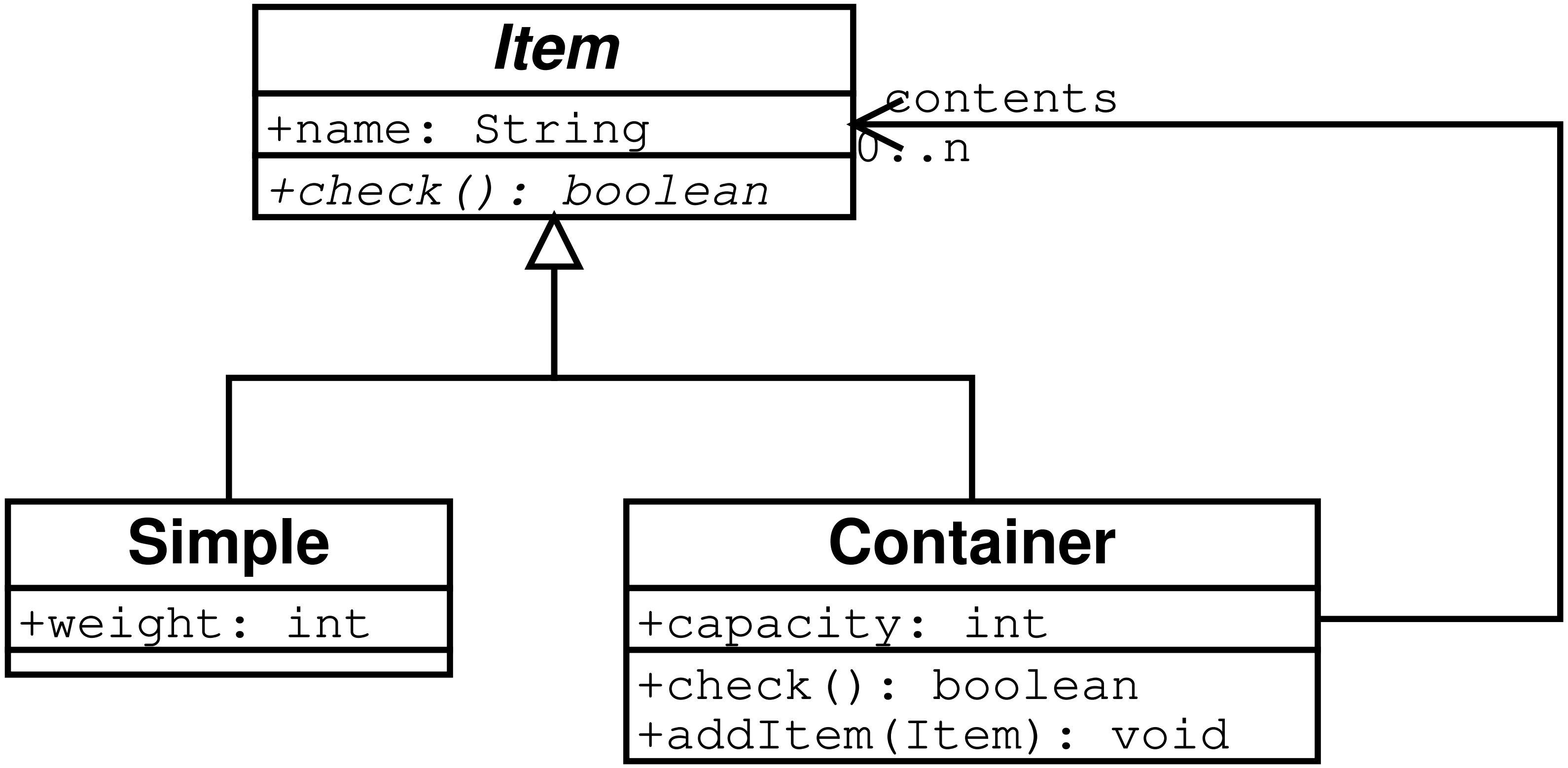
Aspectual Collaborations

Aspectual Collaborations addresses these issues by adding a modular encapsulation to aspects.

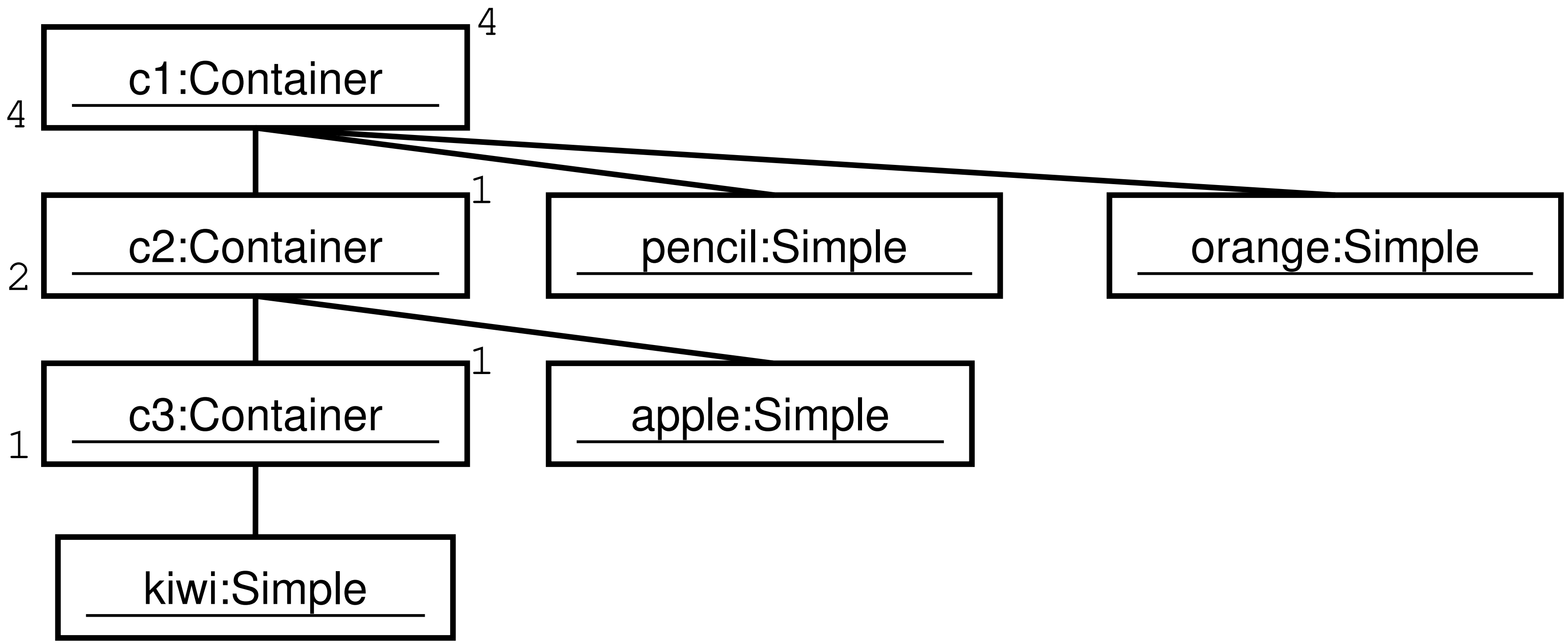
- Flexible reuse.
- Separate compilation.
- Compositional construction.
- Closed set of participant classes, enhanced with ability to have deferred members and aspectual behavior
- Participants generalize java classes.
- Collaborations generalize packages.
- Collaborations composed by point-wise composition of constituent classes

Implementation: acc

Example UML

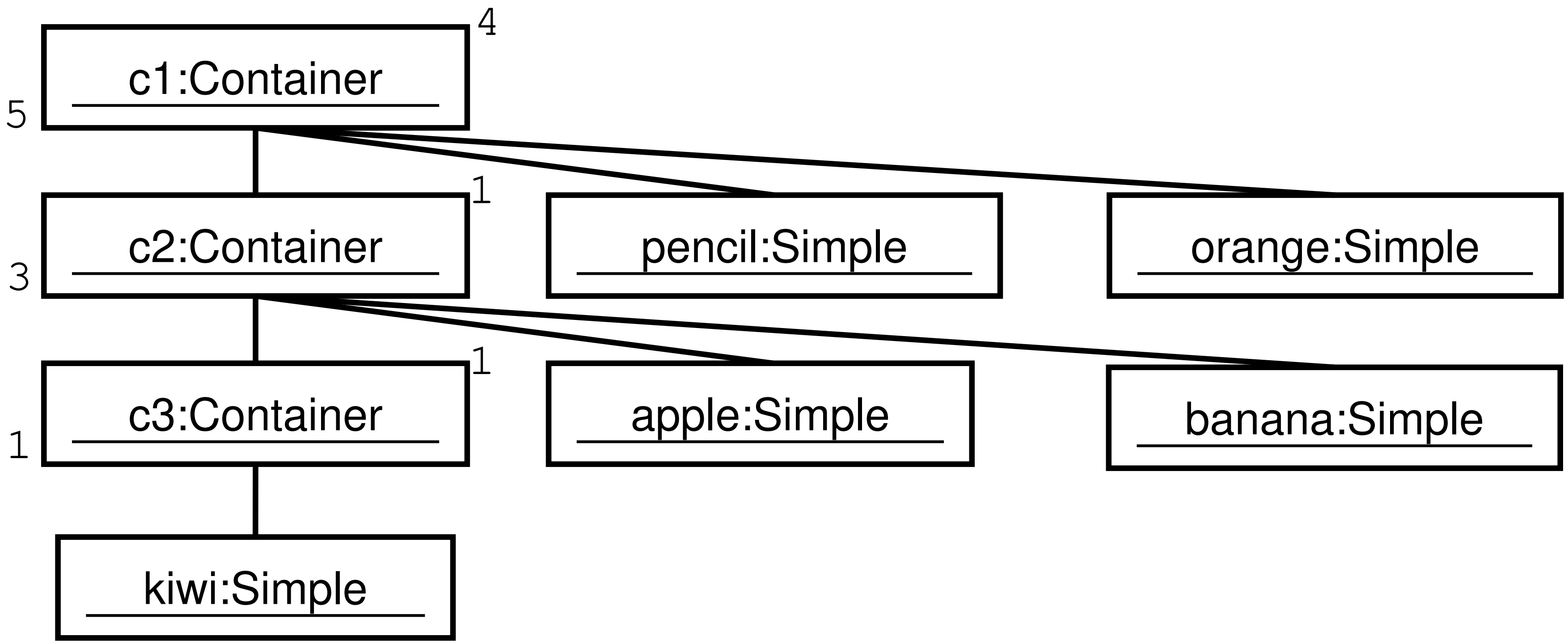

Item
+name: String
+check(): boolean
contents
0..n
Simple
+weight: int
Container
+capacity: int
+check(): boolean
+addItem(Item): void

Example Object Graph



Assume all simples have weight 1. Capacities for containers are in the upper right corner. c1 is OK, c3 is OK, but c2 is overloaded.

Example Object Graph



Adding a banana, we also overload c1, but why recheck c3? Our goal is to write a caching aspect, to avoid this recheck.

Caching behavior requirements

We need to:

- Capture and cache the result of checking a container.
- invalidate this cache when the container or a sub-container is modified.

More precisely:

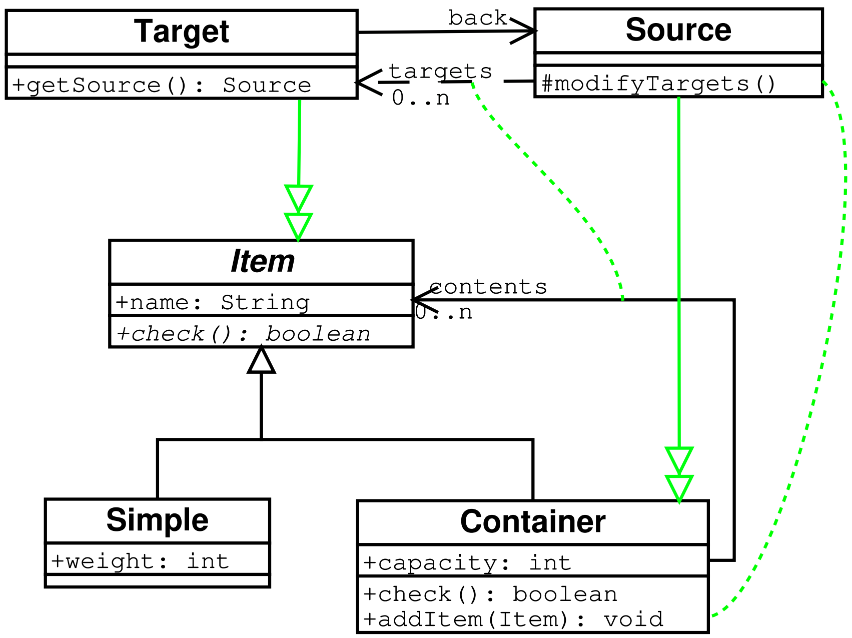
- Add and maintain back-pointer, to know which containers need to be invalidated.
- wrap `check()` in advice to implement caching behavior.
- wrap `addItem()` in advice to invalidate the cache.

And of course, we want it to be done

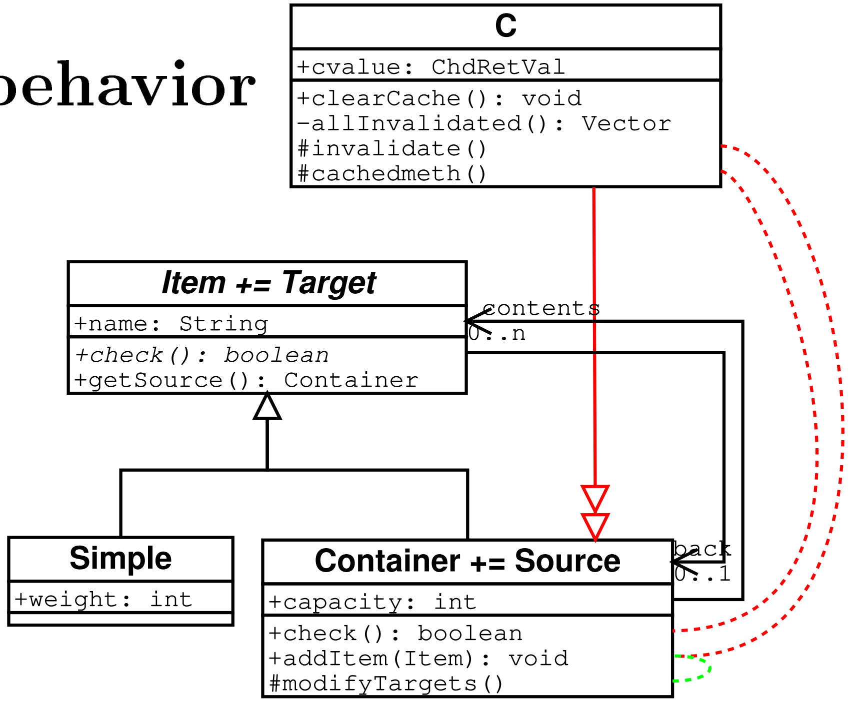
- without modifying the original program (aspectual, eh?)
- without tying the aspect to the original program (reuse!)

The backlink behavior

```
collab backlink;
import java.util.*;
participant Source {
  expected Vector targets;
  aspectual RV modifyTargets(EM e) {{
    RV rv = e.invoke();
    Iterator trgs = targets.iterator();
    while (trgs.hasNext()) {
      ((Target)trgs.next()).back = this;
    }
    return rv;
  }}
}
participant Target {
  Source back;
  Source getSource() {{ return back; }}
}
```



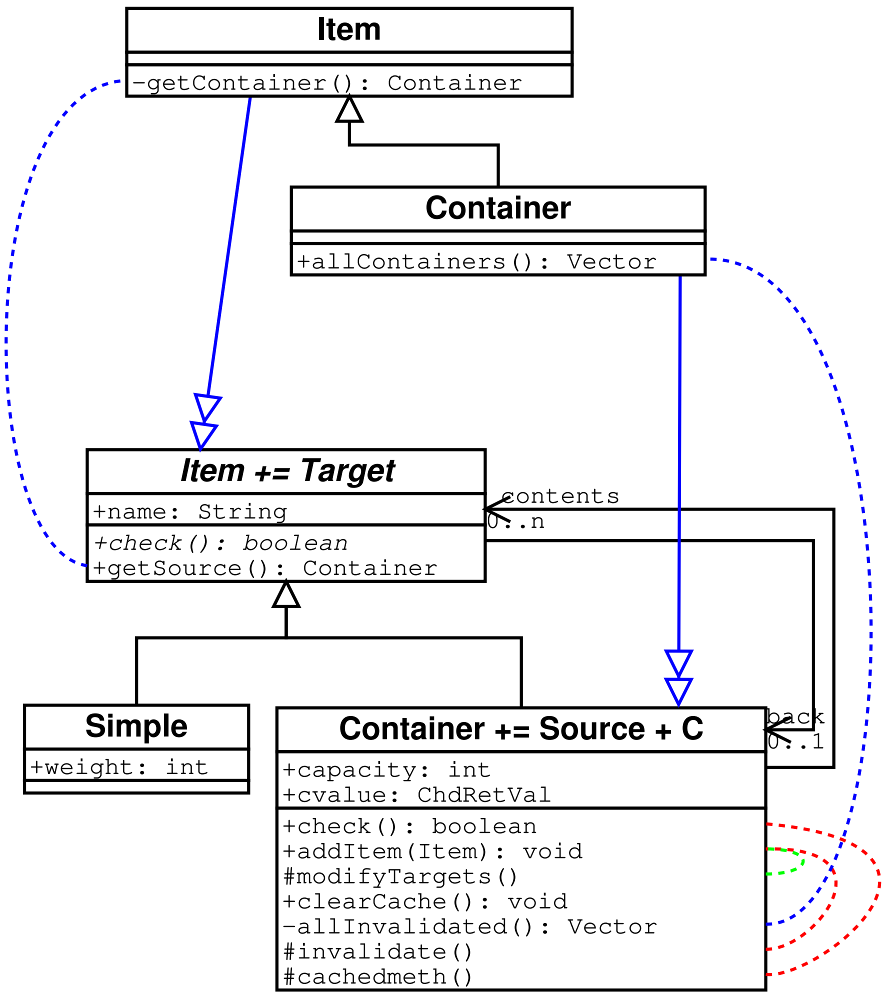
The caching behavior




```
collab caching;
import java.util.*;
participant C {
  ChdRetVal cvalue;
  void clearCache() {{
    System.err.println("clear cache");
    cvalue = null;
  }}
  expected Vector allInvalidated();
  aspectual RV invalidate(EM e) {{
    RV retval = e.invoke();
    Iterator inv =allInvalidated().iterator();
    while (inv.hasNext()) { ((C)inv.next()).clearCache(); }
    return retval;
  }}
  aspectual ChdRetVal cachedmeth(ChdMth e) {{
    if (cvalue==null) { cvalue = e.invoke(); }
    else { System.err.println("using cache"); }
    return cvalue;
  }}
}
```

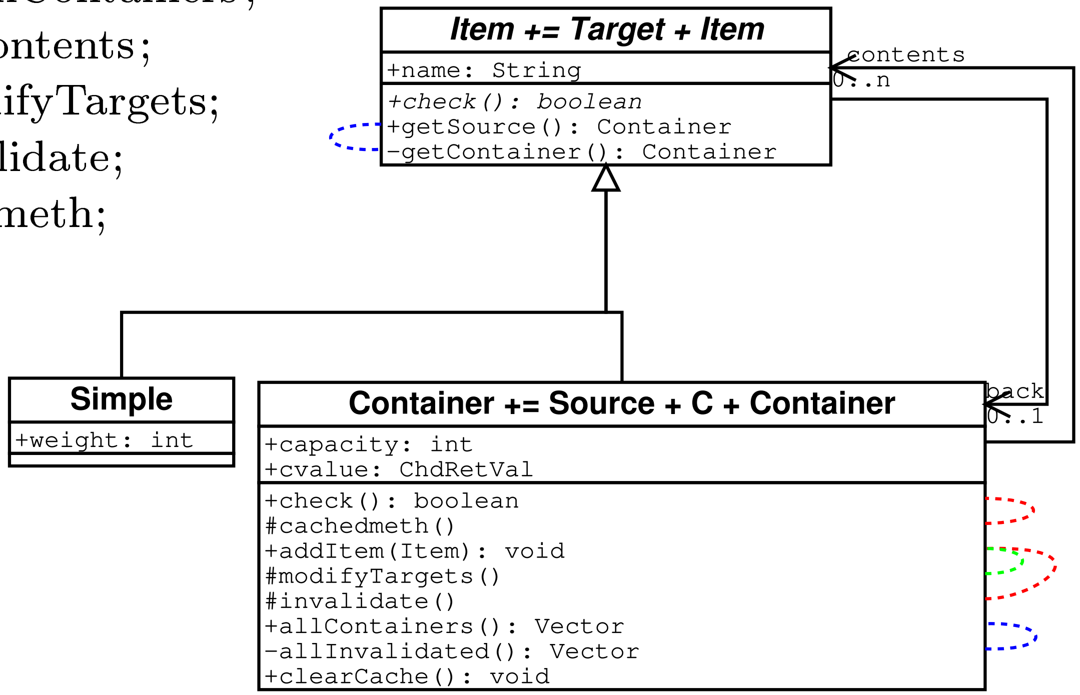
The allcont behavior

```
collab allcont;
import java.util.Vector;
participant Item {
  expected Container getContainer();
}
participant Container extends Item {
  Vector allContainers() {{
    Vector v = new Vector();
    Container c = this;
    while (c != null) {
      v.add(c);
      c = c.getContainer();
    }
    return v;
  }}
}
```



Linking up the result

```
attach backlink, caching, allcont {
  Item += Target, allcont.Item {
    provide getContainer with getSource;
  }
  Container += Source, C, allcont.Container {
    provide allInvalidated with allContainers;
    provide targets with result:contents;
    around result:addItem do modifyTargets;
    around result:addItem do invalidate;
    around result:check do cachedmeth;
  }
}
```



Conclusion

- Each collaboration consists of participant classes written over a well defined interface.
- Collaborations are composed by pairwise combination of participant classes.
- Imports, exports, and aspectual behavior are all achieved by playing with member references in the combined participants.

The End

Backup slides beyond this point.

Differences to AspectJ

Differences to HyperJ

Differences to Units