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.

Representative examples: HyperJ, AspectJ.

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 linking language

Aspectual Collaborations

Address these issues by adding a modular encapsulation to aspects.

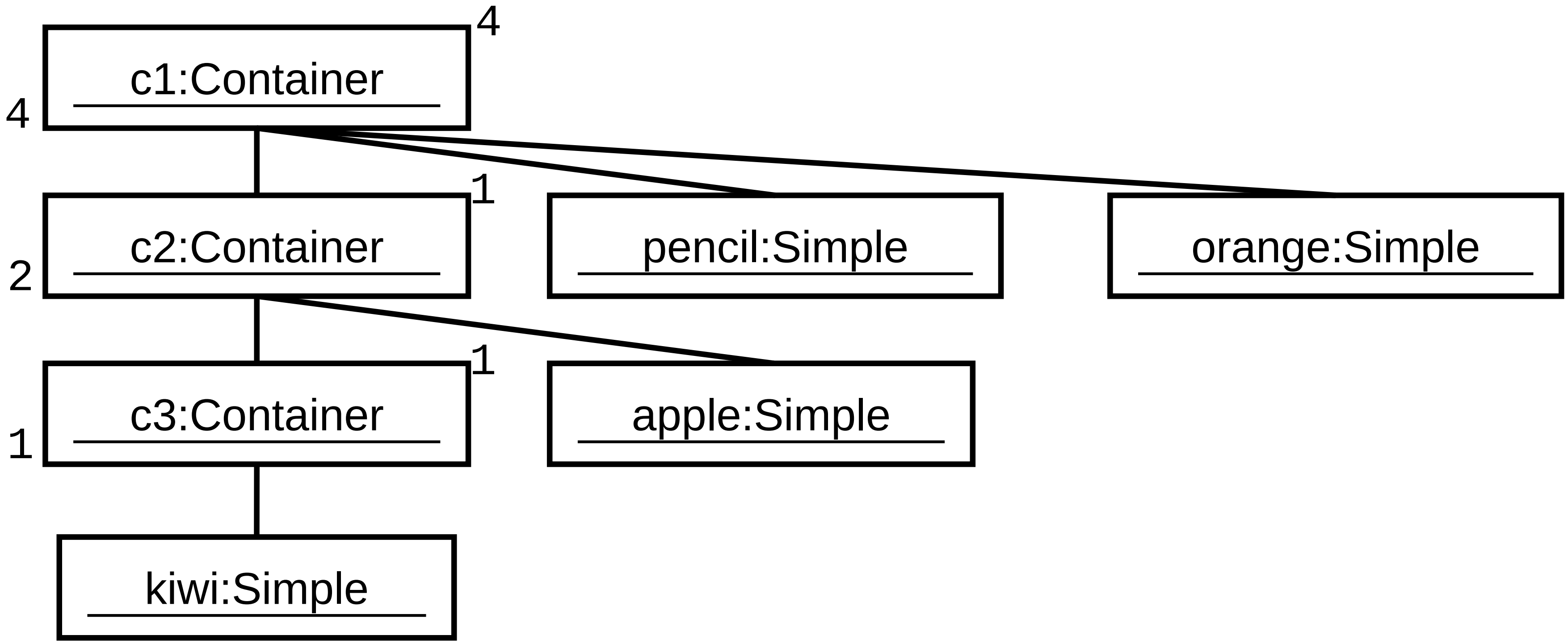
- Closed set of participant classes, enhanced with ability to have deferred members and aspectual behavior
- Participants generalize Javaclasses.
- Collaborations generalize packages.
- Collaborations composed by point-wise composition of constituent classes

This achieves:

- Flexible reuse.
- Separate compilation.
- Compositional construction.

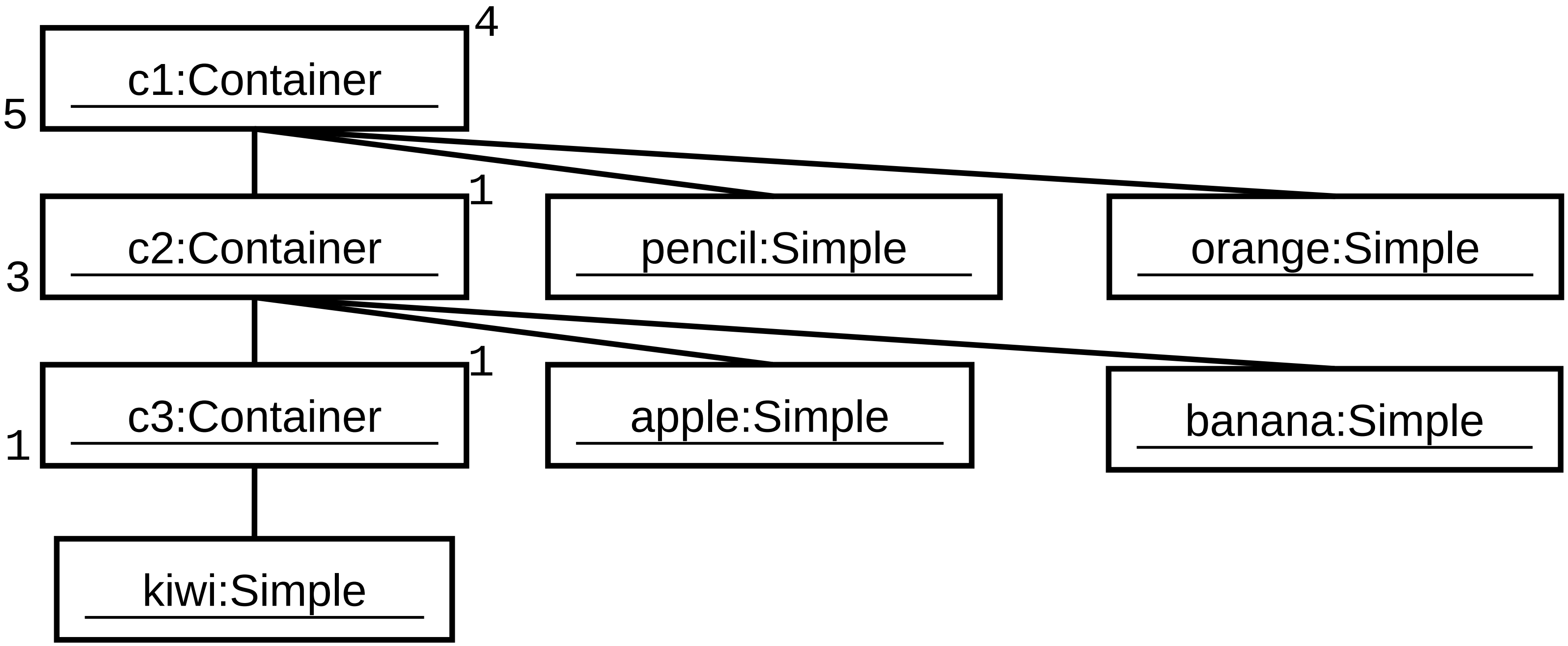
Prototype Implementation: acc

Example



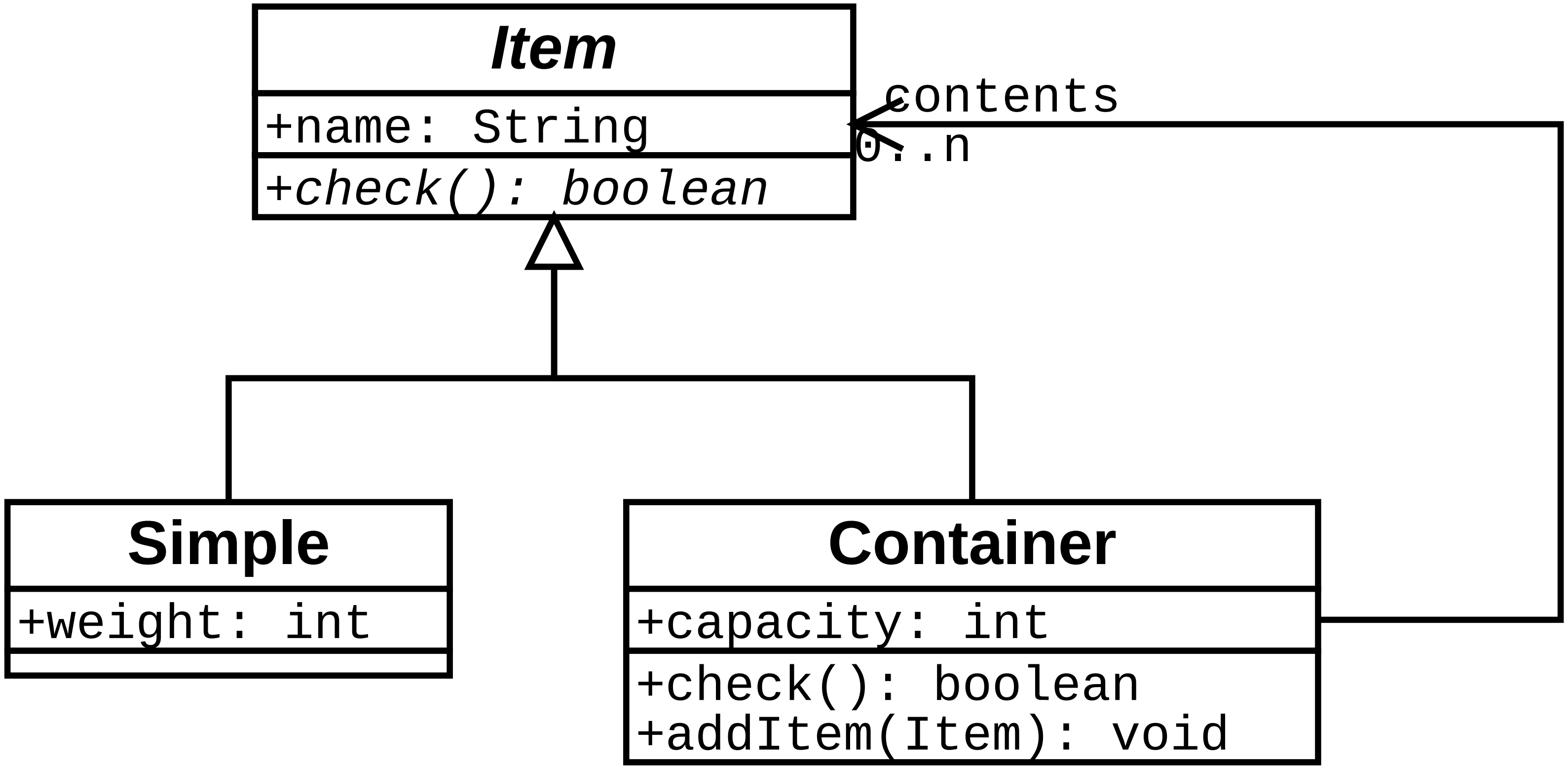
Assume all simples have weight 1. Capacities for containers are in the upper right corner. c1 is OK, c3 is OK, but c2 is overloaded.

Example



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

Example UML


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

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:

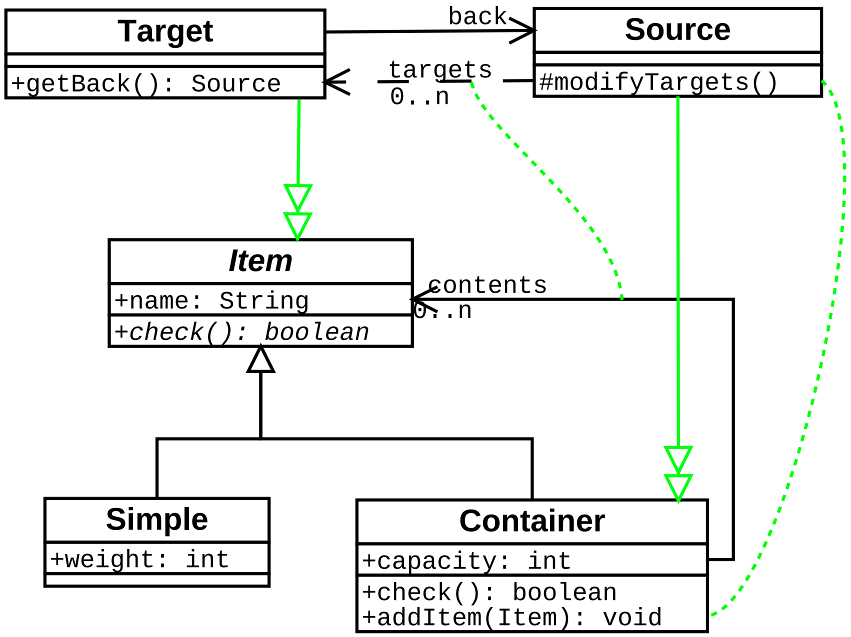
- Add and maintain a contained-in, to know which containers need to be invalidated.
- wrap `check()` in advice to implement caching behavior.
- wrap `addItem()` in advice to invalidate the cache.

And of course, we want it to be done

- without modifying the original program (aspectual)
- without tying the aspect to the host program (reuse)

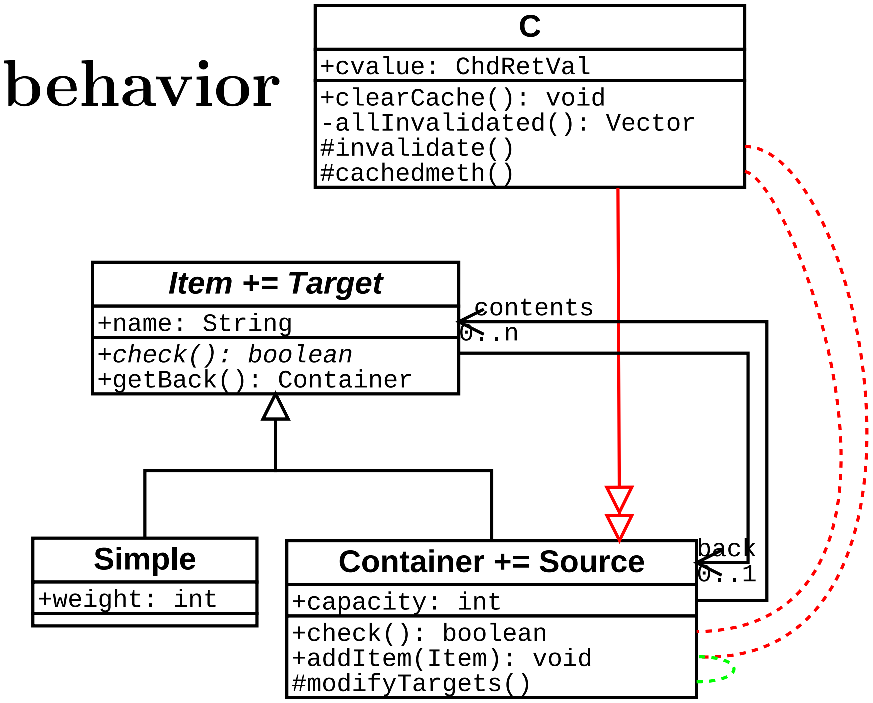
The backlink behavior

```
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 getBack() {{ return back; }}
}
```



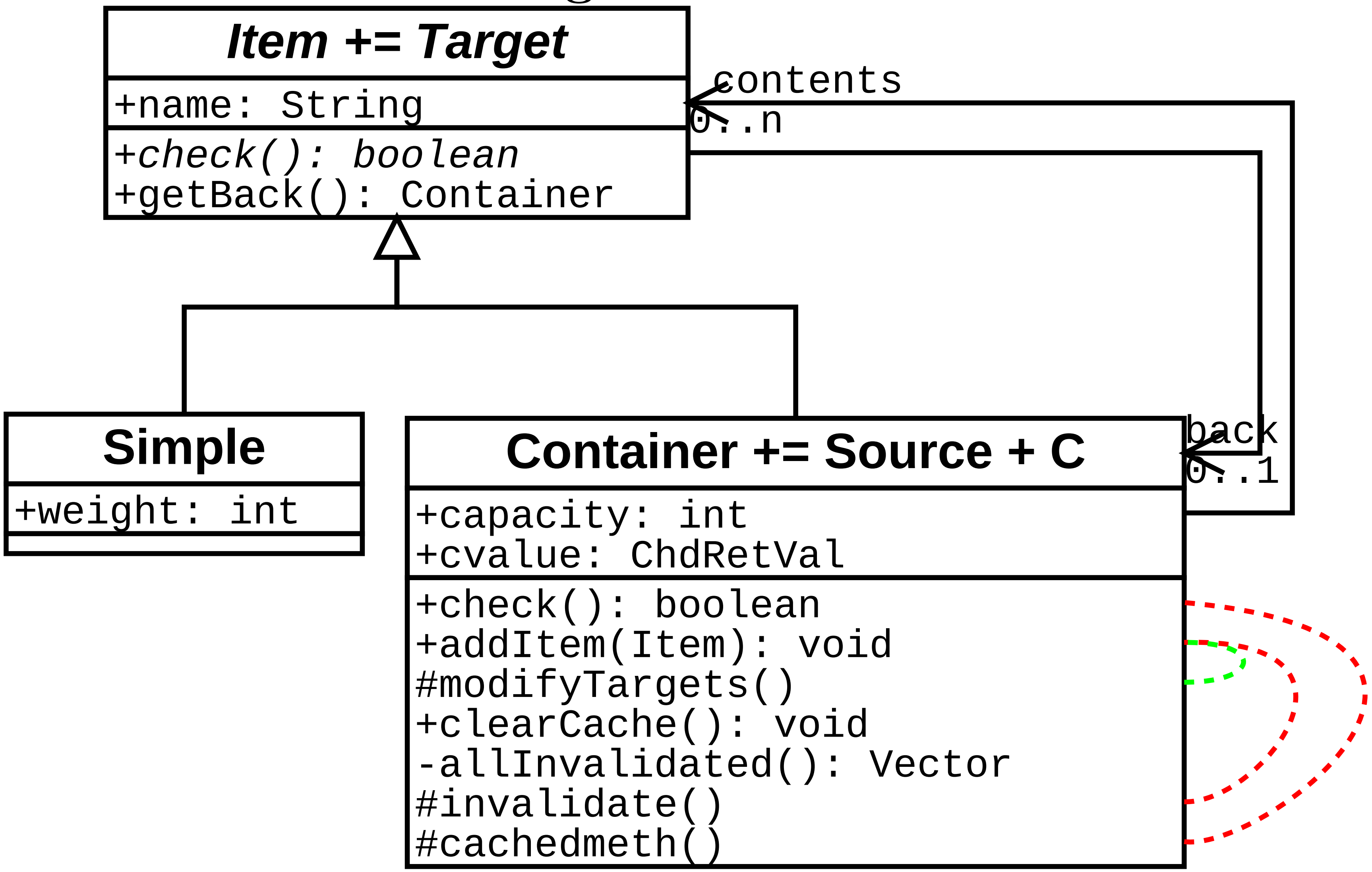
The backlink collaboration expects 1) an association (vector) from Source to Targets, and 2) some method that modifies this association. From these it ensures that each Target has a backlink to the source

The caching behavior



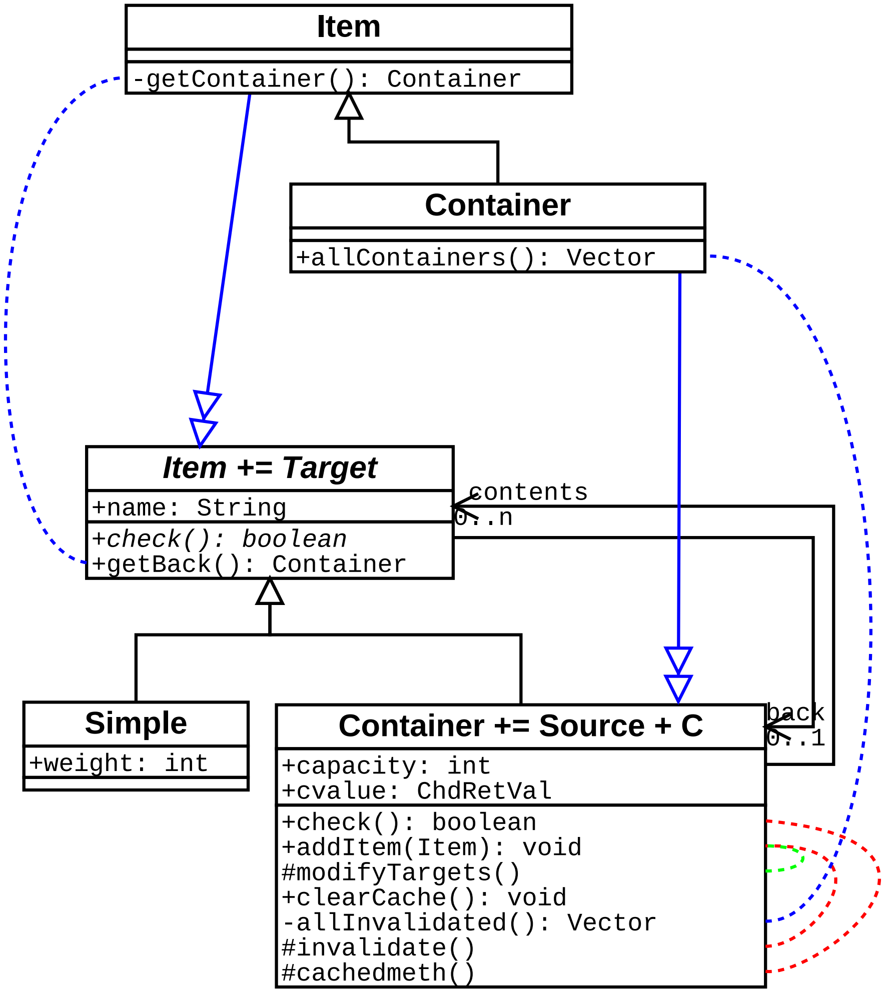
```
collab caching;
import java.util.*;
participant C {
  ChdRetVal cvalue;
  void clearCache() {{
    System.err.println("clear␣cache");
    cvalue = null;
  }}
  expected Vector allInvalidated();
  aspectual RV invalidate(EM e) {{
    RV retval = e.invoke();
    Iterator inv =allInvalidated().iterator();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

After inserting cache and backlink



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;
  }}
}
```



Item
-getContainer(): Container

Container
+allContainers(): Vector

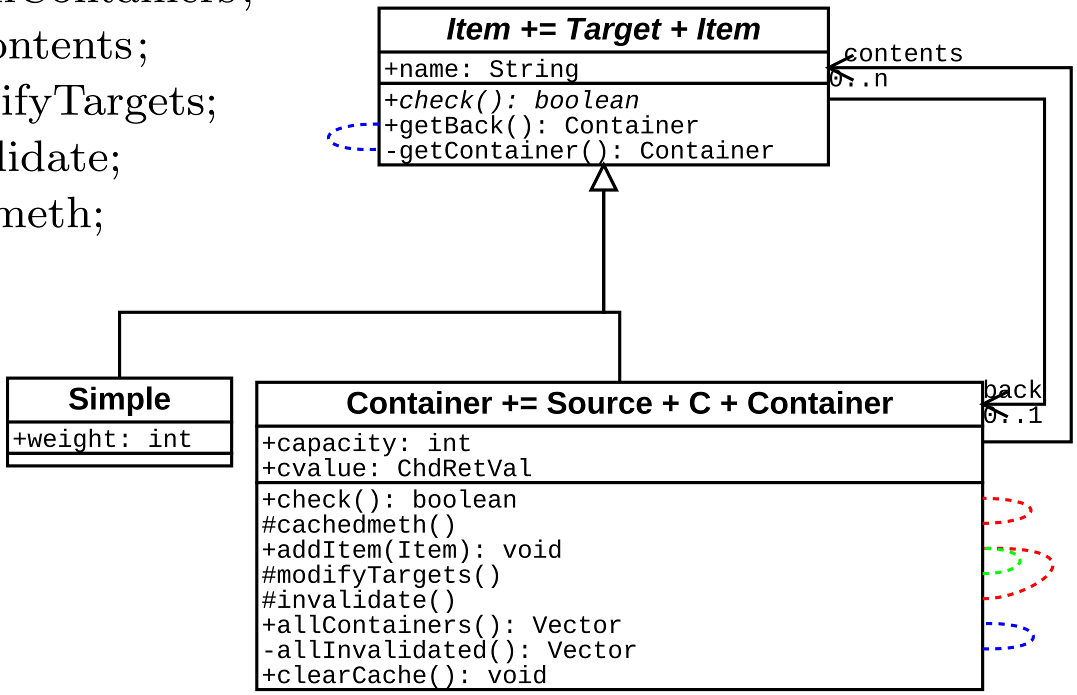
Item += Target
+name: String
+check(): boolean
+getBack(): Container

Simple
+weight: int

Container += Source + C
+capacity: int
+cvalue: ChdRetVal
+check(): boolean
+addItem(Item): void
#modifyTargets()
+clearCache(): void
-allInvalidated(): Vector
#invalidate()
#cachedmeth()

Linking up the result

```
attach backlink, caching, allcont {
  Item += Target, allcont.Item {
    provide getContainer with getBack;
  }
  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;
  }
}
```



Item += Target + Item

+name: String

+check(): boolean
+getBack(): Container
-getContainer(): Container

Simple

+weight: int

Container += Source + C + Container

+capacity: int
+cvalue: ChdRetVal

+check(): boolean
#cachedmeth()
+addItem(Item): void
#modifyTargets()
#invalidate()
+allContainers(): Vector
-allInvalidated(): Vector
+clearCache(): void

Conclusion

We have demostrated a simple system which attempts to combine aspectual programming with a module system.

- We are able to program (and separately compile) aspectual behaviors.
- The behaviors are written against their own class graph interface, with “holes” to plug in attachment specific behaviors.
- The aspectual collaborations are composed by pointwise class insertion, creating a collaboration with hopefully fewer “holes”.
- When all holes are filled, we have (potentially) runnable application. Of course composition can continue further.
- By varying attachment details, the same collaboration can be reused in different ways in the same application.

The End

Backup slides beyond this point.

What we haven't told you about

Features

- Exported vs unexported members
- Matching and multiple attachments
- Sharing between multiple attachments
- Accessing arguments and return values to aspectual methods

Futures

- Self hosting
- Object Graph constraints
- Refinement between collaborations
- Parametric Collaborations
- We may be able to be more flexible w.r.t. mimicking class structure in allcont.

Difficulties

- Constructors
- Wrapping and providing overrid(den/ing) members

Differences to AspectJ

- Separate Compilation
- Encapsulation
- JPM : we only have member defintion/invocation as join point

Differences to HyperJ

- Cannot do post-hoc remodularisation – not without either wasting alot of space or implementing dead def removal.
- Shares idea of inserting code into classes to compose.
- Have more flexible combinators than Hyper/J

Differences to Units

- Binding time; we are inherently early, but with funky linking language. Units bind classnames late. Some of the programming patters units use are applicable to collaborations as well.
- Use inheritance rather than insertion
- Overriding should be able to get some aspectual benefits. Would need program generator to do the generic aspectual stuff.