

## Applying the Universe Type System to an industrial application

Case Study

Thomas Hächler

Master Project

September 2004 - March 2005

Supervising Assistant: Dipl.-Ing. Werner M. Dietl

Supervising Professor: Prof. Dr. Peter Müller



Barcode based ticketing system.

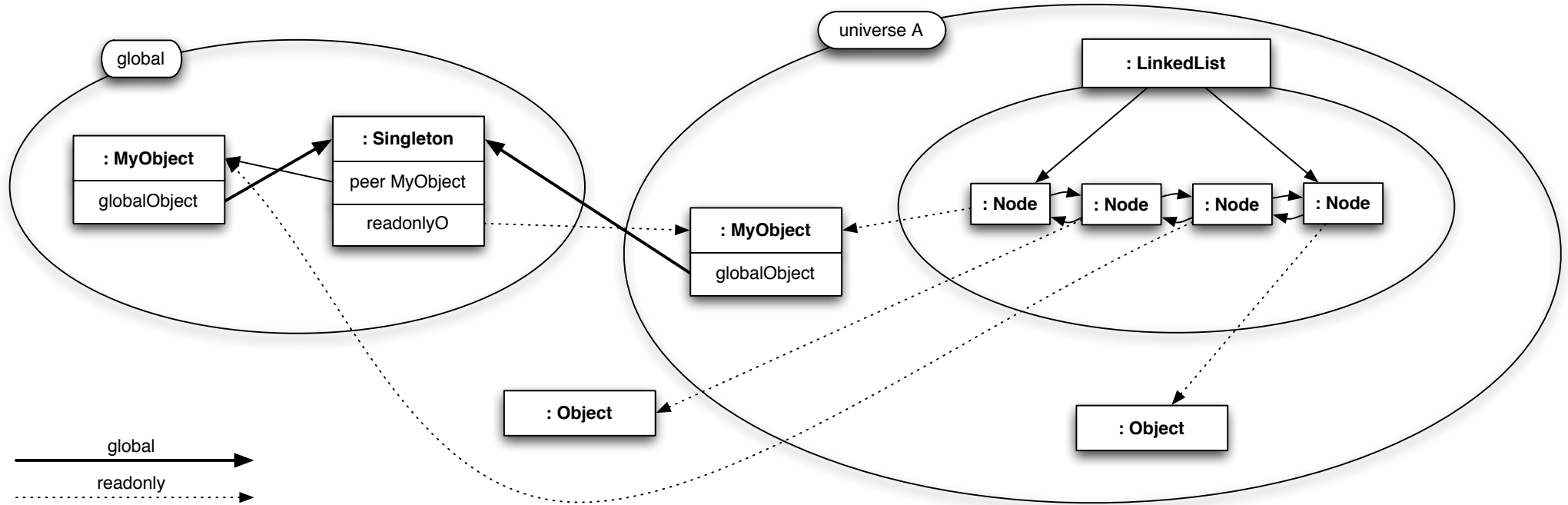
Runs on device called "Yoshi".

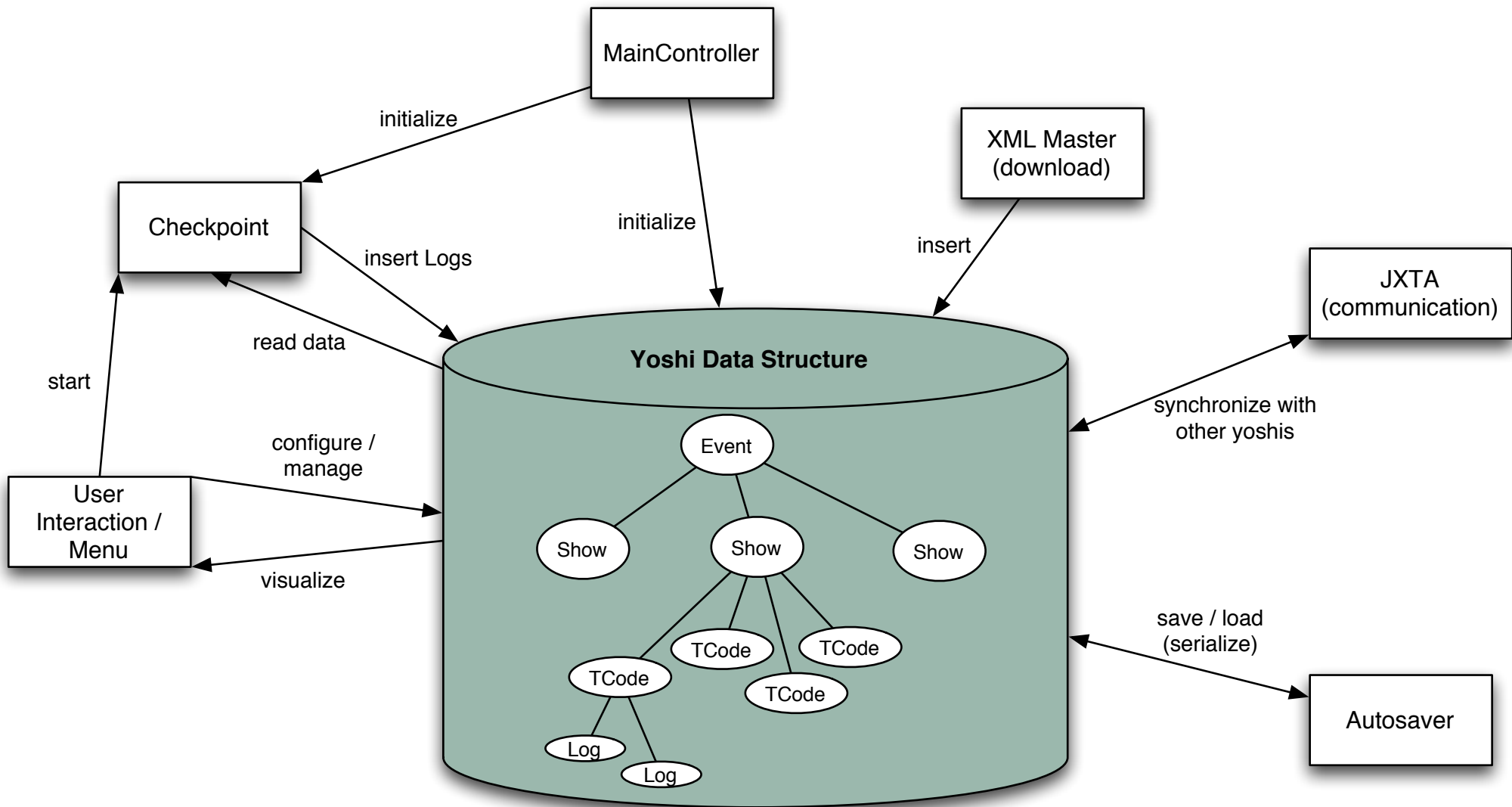
Checks print at home<sup>®</sup> tickets bought on [www.starticket.ch](http://www.starticket.ch).

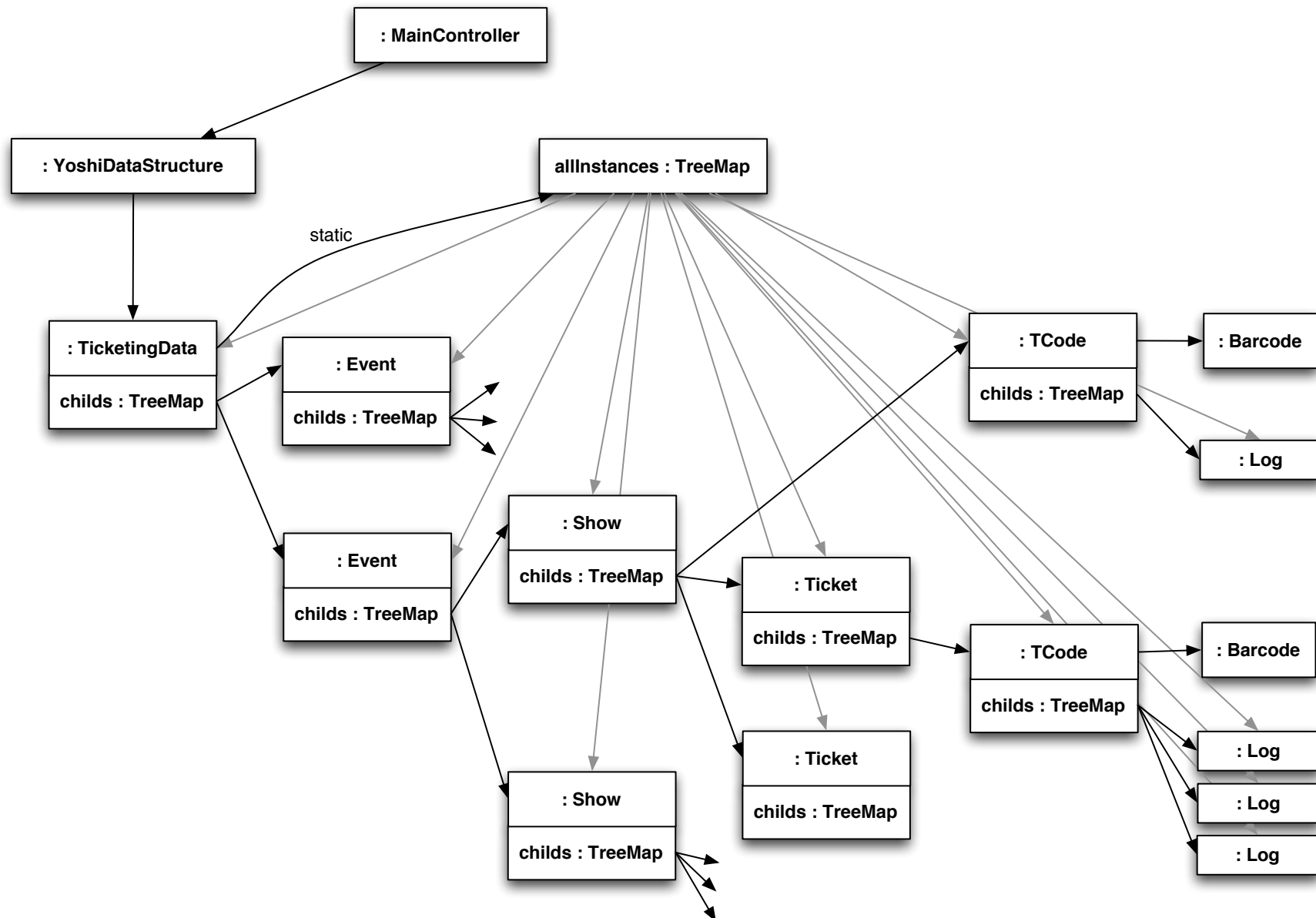
- Introduction to the Universe Type System
- Introduction to the application
- Real-world experiences
- Applying the Universe Type System to Java API
- Proposals to face encountered problems
- Conclusion

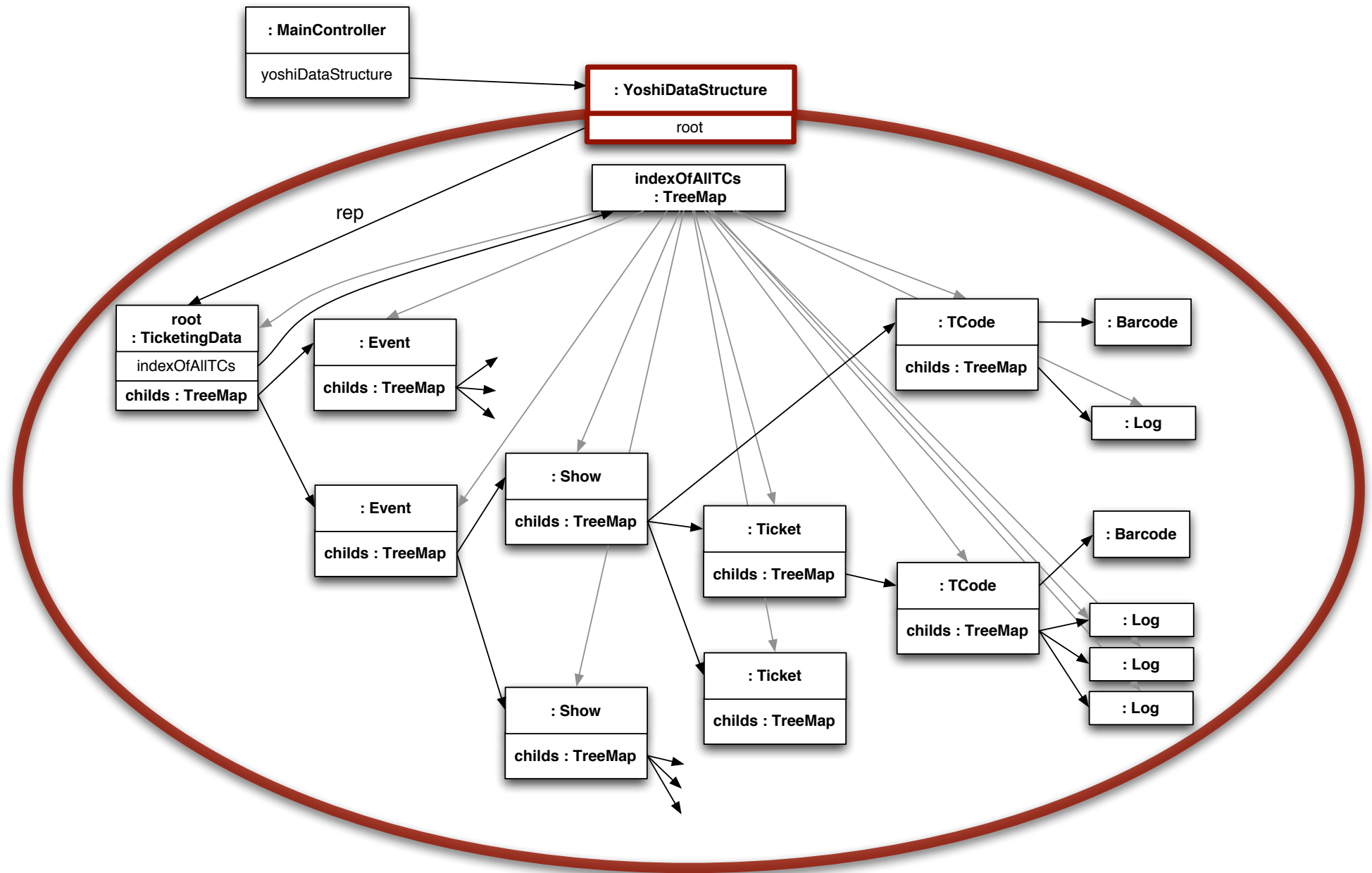
The Universe Type System structures the object store.

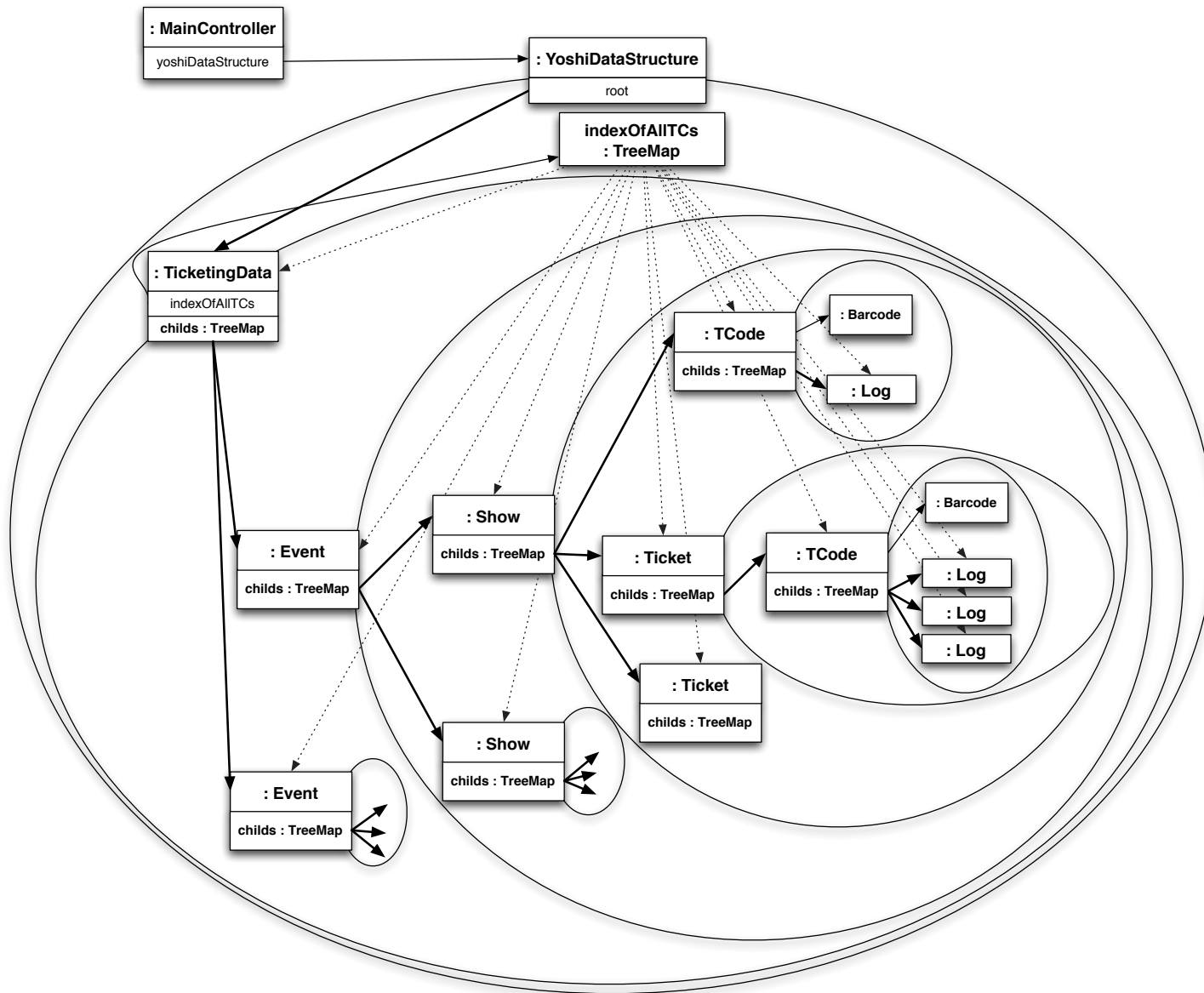
- Ownership relations that define the universes.
- `rep`, `peer` and `readonly` references
- A type system guarantees the defined properties at compile time.
- Extension needed: `global` universe

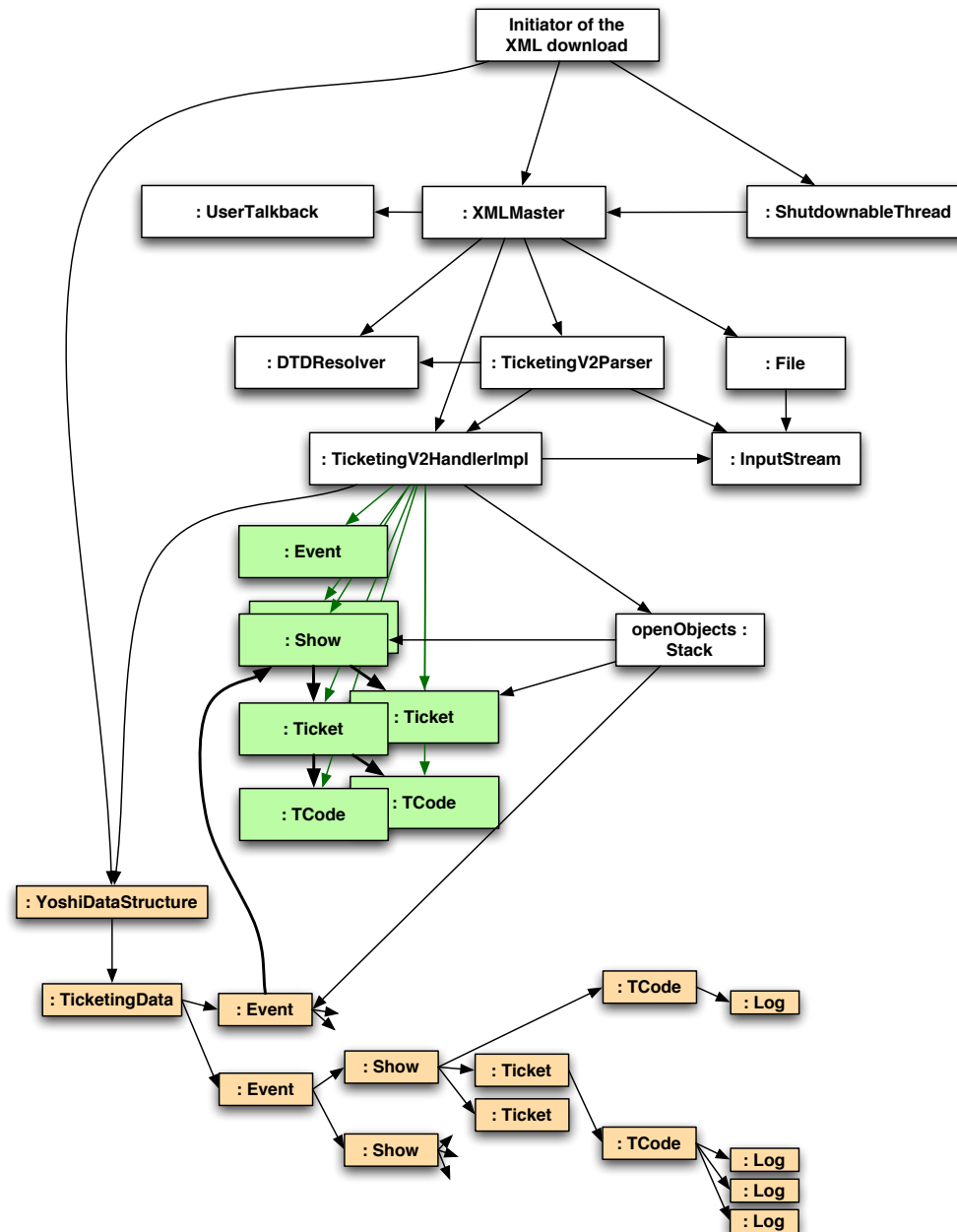


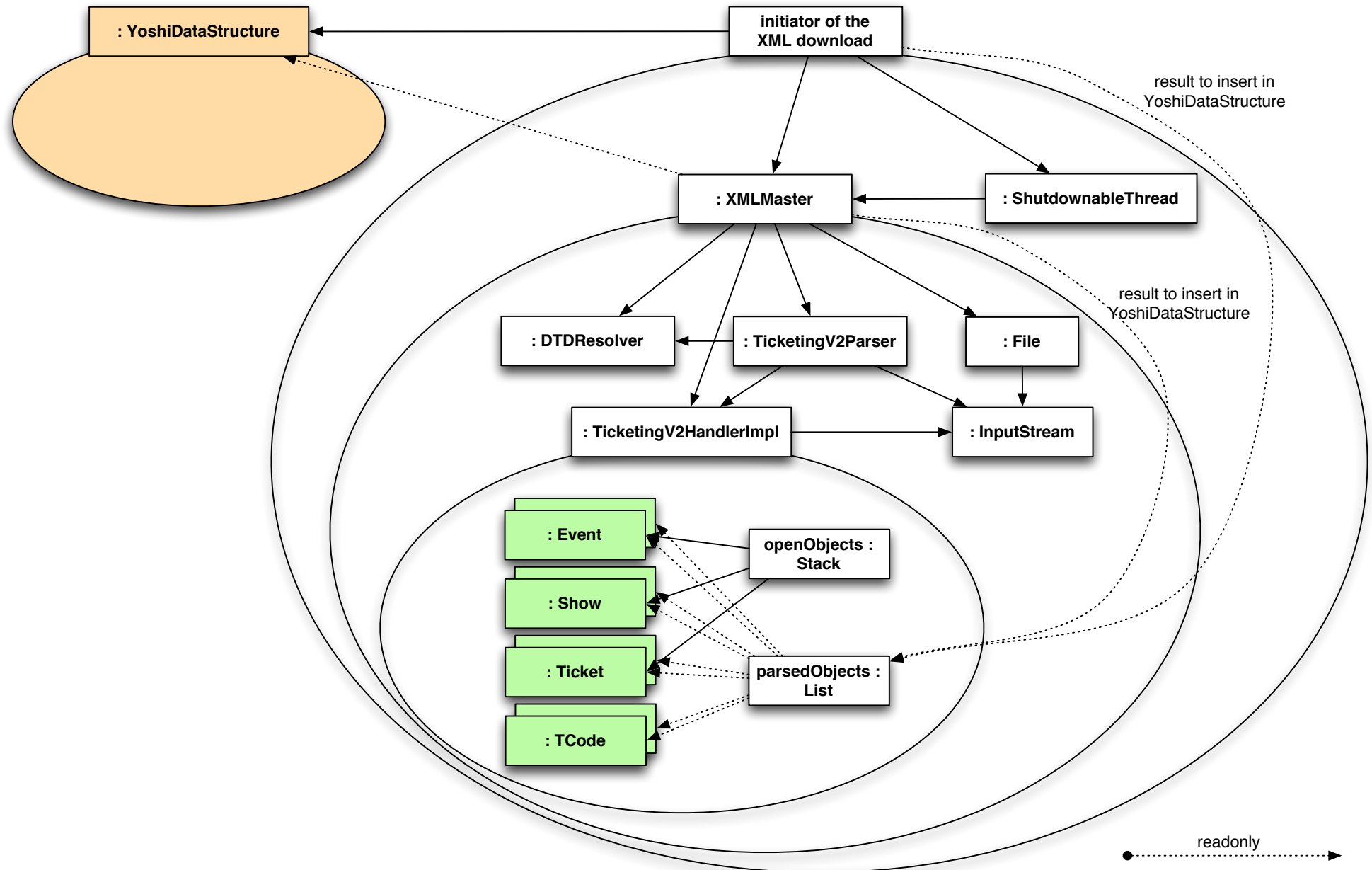








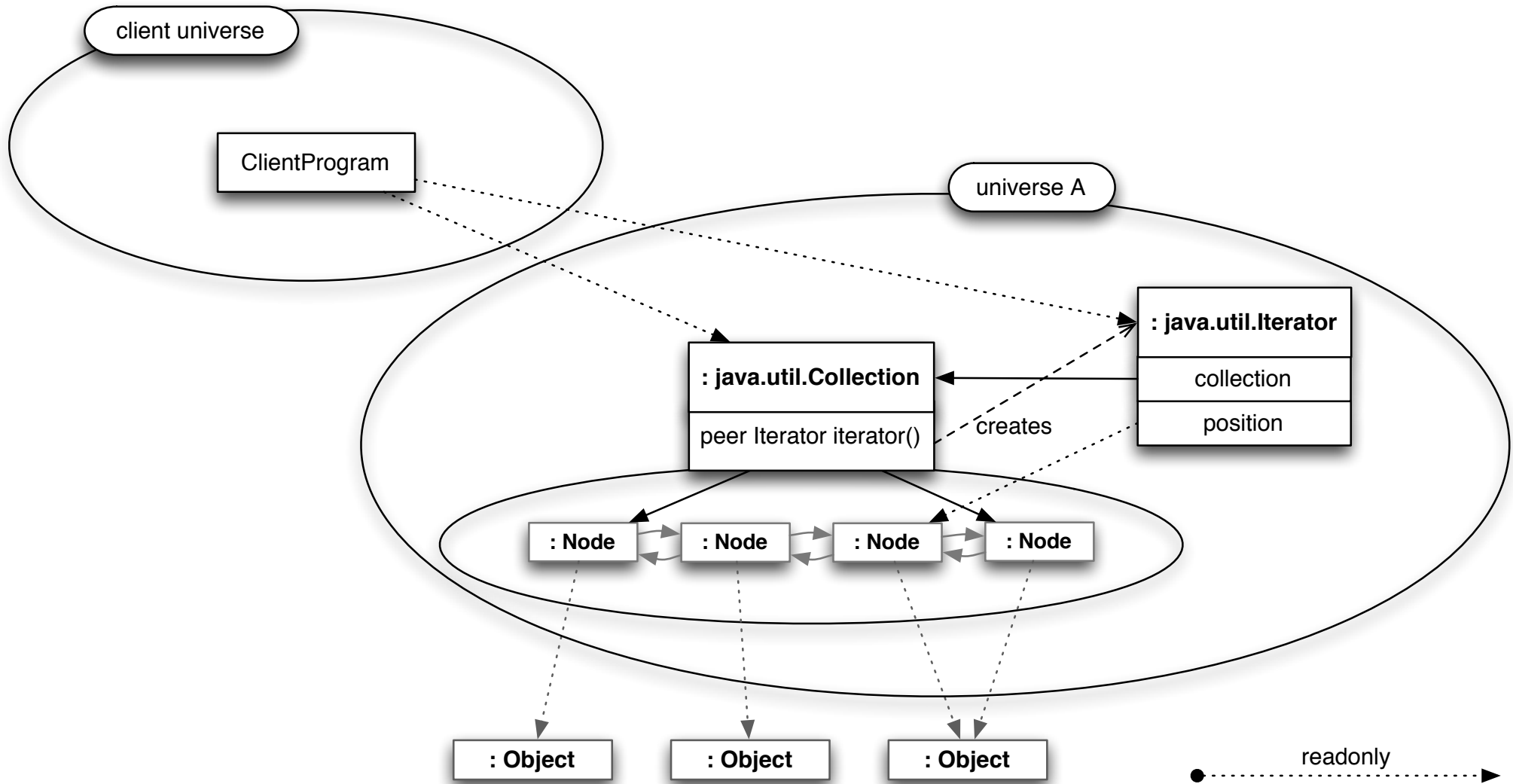


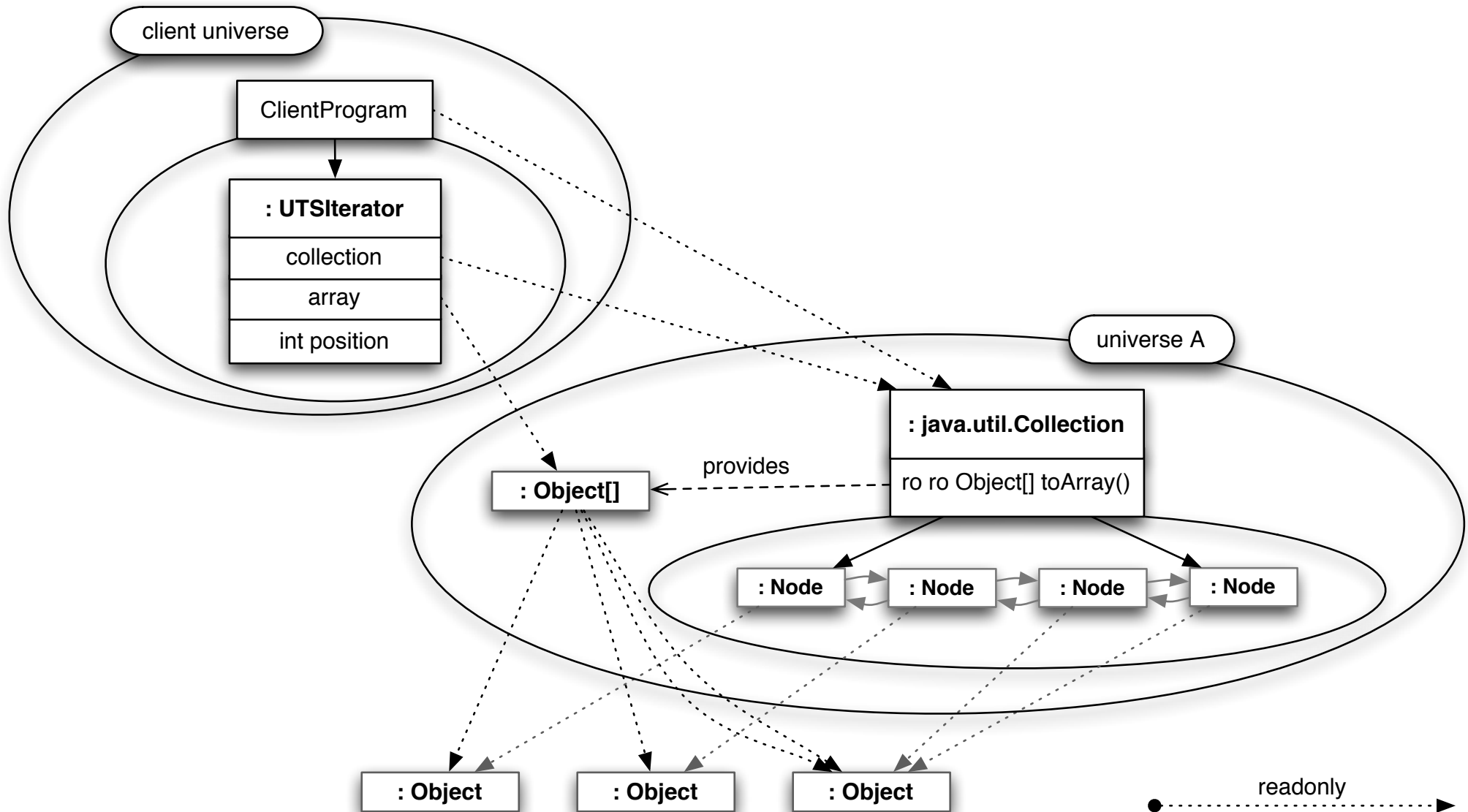


- In which universe should `System.out` be?
- Should it be possible to iterate over a `readOnly` collection?
- How should the parameter `b` in `InputStream.read(byte[] b)` be annotated?

- In which universe should `System.out` be?

```
public static global PrintStream out;
```





in `InputStream` (and other `java.io.*`):

```
InputStream.read(peer byte[] b);  
InputStream.read(rep byte[] b);  
InputStream.read(global byte[] b);
```

in `StringBuffer`:

```
void getChars(int srcBegin, int srcEnd, peer char[] dst, int dstBegin);  
void getChars(int srcBegin, int srcEnd, rep char[] dst, int dstBegin);  
void getChars(int srcBegin, int srcEnd, global char[] dst, int dstBegin);
```

in a composite pattern:

```
abstract void collectInformation(peer Map m);  
abstract void collectInformation(rep Map m);  
abstract void collectInformation(global Map m);
```

```
class C {  
    <writable U> void writeTo(U PrintStream p) {  
        p.println("hello _universe!");  
    }  
}
```

*// client with peer reference c to instance of C:*

```
peer C c = new peer C();
```

```
c.writeTo(System.out);
```

*// U is resolved to be global.*

```
c.writeTo(new peer PrintStream(..));
```

*// U is resolved to be peer.*

```
c.writeTo(new rep PrintStream(..));
```

*// U is resolved to be readonly*

*// => incompatible => compile time error.*

- `writable` stands for { `peer`, `rep`, `global` }
- `writable` allowed for formal parameters, local variables and return values only
- Not allowed for fields – Restrictions for actual parameters
- Compiler checks whether the actual parameter is read-write for the callee

- Lot of work until application compiled with MJ/JML
- Bug reports for MJ/JML compiler
- `Copyable` – an interface to copy objects crossing universe boundaries
- Proposal for implicit `readonly` annotation
- Proposal for method-local universes
- Workarounds for programming patterns

- Universe Type System works
- Universe Type System implies better structures
- Restructuring required – runtime overhead possible
- `global` universe needed in real world (logging, properties, singletons)
- Restructuring of API needed
- Some ideas to make life easier

```
<writable U> void writeTo(U PrintStream p) {  
    p.writeln("hello_universe!");  
}
```

```
void writeToPeer(peer PrintStream) { .. /* [ U / peer ] */ }  
void writeToRep(rep PrintStream) { .. /* [ U / rep ] */ }  
void writeToPeer(global PrintStream) { .. /* [ U / global ] */ }
```

```
void writeTo(readonly PrintStream p) {  
    if (p instanceof peer PrintStream) {  
        writeToPeer((peer PrintStream) p); return; }  
    if (p instanceof rep PrintStream) {  
        writeToRep((rep PrintStream) p); return; }  
    if (p instanceof global PrintStream) {  
        writeToGlobal((global PrintStream) p); return; }  
    throw new ClassCastException("p_is_not_writable");  
}
```

```
/**
 * It is recommended to implement a constructor of the following form:
 * <code>MyClass(MyClass o) { copyFrom(o); }</code> */
public interface Copyable {
    /**
     * This method takes another Object of the same type
     * and copies its internal state to this.
     *
     * implemenation of sheep-copy is recommended:
     * - new Objects for rep- and peer-references (sheep-copy as well).
     * - copy the readonly-references and the values. */
    void copyFrom(readonly Copyable o) throws ClassCastException;
}

class MyClass implements Copyable {
    /** recommended constructor */
    MyClass(readonly MyClass o) { copyFrom(o); }
    void copyFrom(readonly Copyable o) {
        // nothing to do in the case of no instance fields.
    }
}
```

```
void insertInChild(readonly Component comp) throws Exception {
    readonly Composite roParent = comp.parent();
    while (!this.equals(roParent.parent())) {
        if (roParent == null) throw new Exception("parent_of_" + comp + "_not_
            found.");
        roParent = roParent.parent();
    }
    // cast roParent to rep.
    rep Composite repParent = (rep Composite) roParent;
    if (repParent.equals(comp.parent())) {
        repParent.insert(comp);
    } else {
        repParent.insertInChild(comp);
    }
}

/** replaces a possible old entry with same key */
void insert(readonly Component roComp) {
    // Component implements Copyable
    // children: field of type java.util.Map
    this.children.put(roComp.getKey(), new rep Component(roComp));
}
```

```
import java.util.Iterator;
/**
 * The basic idea is to provide an Iterator over a readonly collection.
 * So an implementation of this interface should provide a constructor with
 * a parameter of type readonly { @link java.util.Collection}</code>.
 */
public interface UTSIterator extends Iterator {

    pure boolean hasNext();

    readonly Object next();

    /**
     * An implementation of this method normally throws an { @link
     * UnsupportedOperationException}
     * because the underlying collection is readonly</code>
     * (and therefore cannot be modified, like { @link #remove()} would do it).
     */
    void remove() throws UnsupportedOperationException;
}
```