Creational Patterns

From Gamma et al.

Behavioural, Structural, **Creational** Patterns

- Recall that patterns fit broadly into three categories: Behavioural, Structural, and Creational.
- **Creational** patterns focus on how objects are created
- Next we look at two creational patterns: **Factory Method** and **Builder**.

Factory Method Pattern

- Idea: "Let subclasses decide which class to instantiate".
- Motivating example.
  - We want to design a text editor framework that can edit a variety of document types (Rich Text Format, HTML, plain old text, etc.)
  - In swing.text the objects that make up a text editor know an EditorKit object.
  - Implementing the "new" menu item, we ask the EditorKit object to create a new document.

Old fashioned solution

- One way to do this involves a single EditorKit class. In EditorKit we have:

```
// Constructor
EditorKit( String textType ) { this.textType = textType ; }

// Factory Method
Document makeDocument() {
    if( textType.equals( "rtf" ) {
        return new RTFDocument() ; }
    else if( textType.equals( "html" ) {
        return new HTMLDocument() ; }
    … }
```

- Problem: To add new document kind, we must edit this class. Thus it is not reusable.

Factory Method Solution

- EditorKit is an abstract class with method

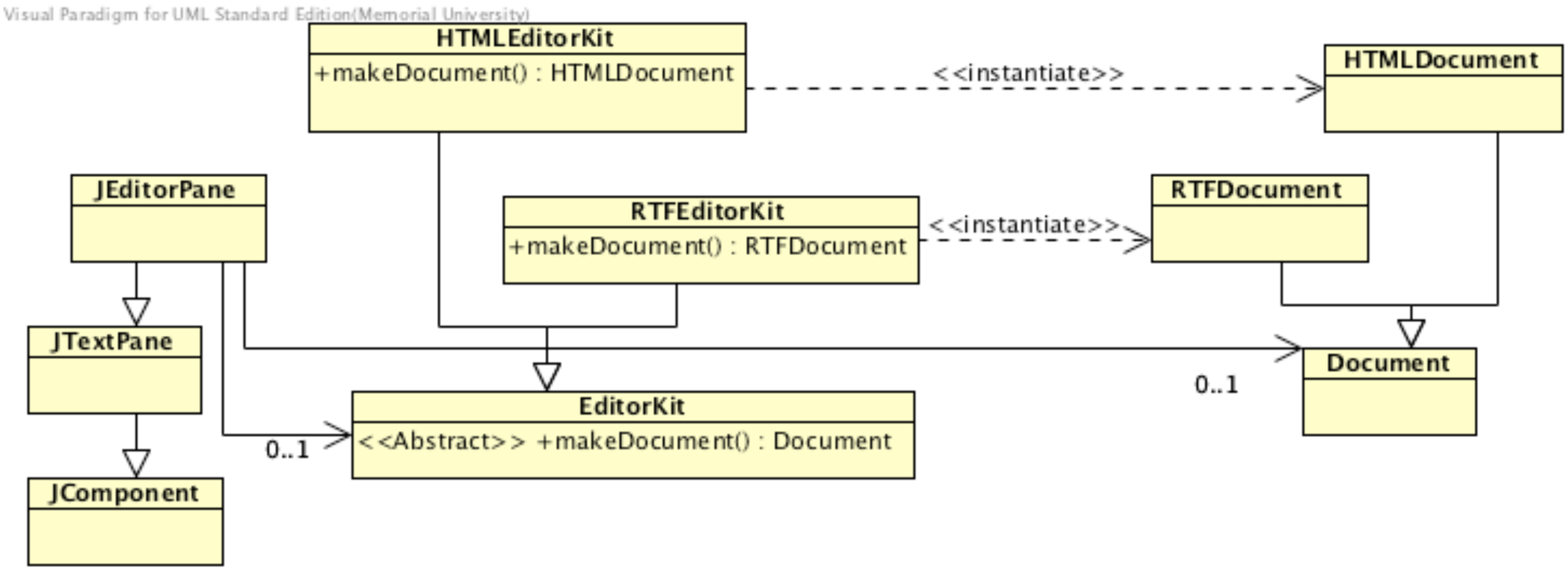
```
// Factory Method
abstract Document makeDocument() ;
```

- EditorKit has a number of subclasses

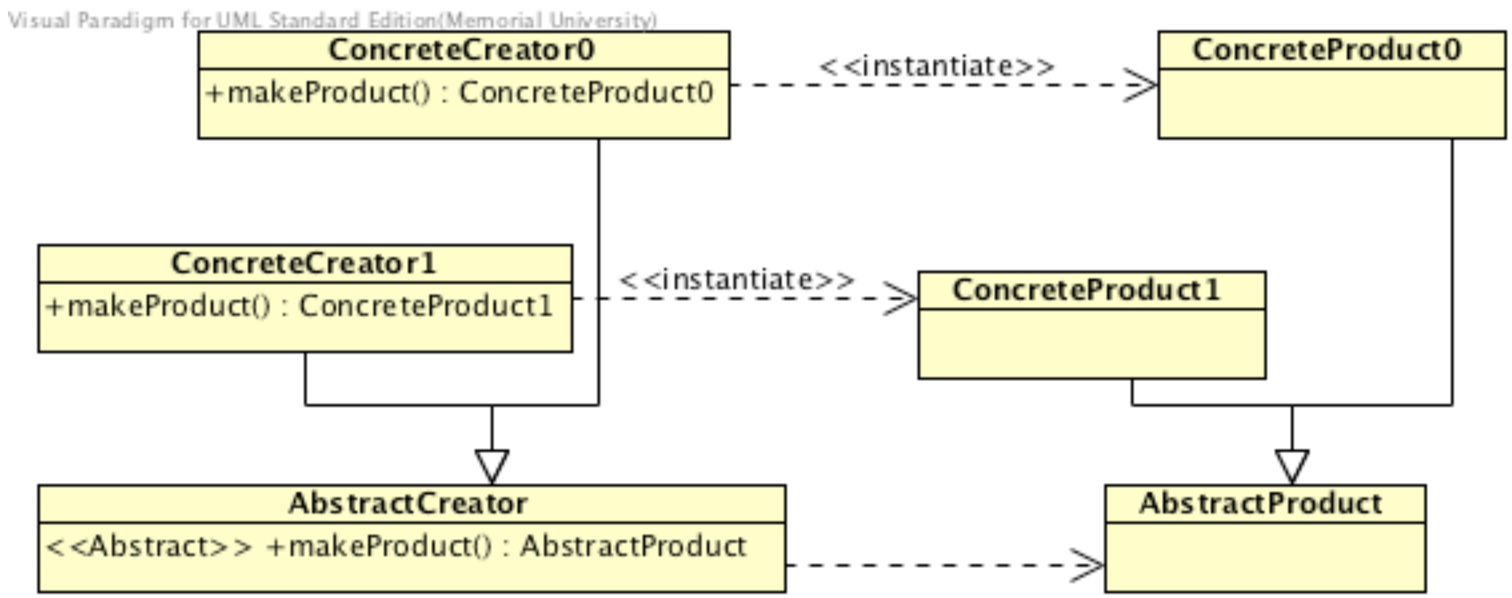
```
class RTFEditorKit extends EditorKit {
    …
    // Factory Method
    Document makeDocument() { new RTFDocument() ; }
```

- Adding a new document type means creating a new subclass of Document and a new subclass of EditorKit. It does not require editing EditorKit, Document, or the code of the text editor.
- EditorKits are factories for Documents

Class Diagram



Class Diagram for Factory Pattern



Factory Pattern in the Collection classes

- Package java.util defines a number of
  - interfaces (Collection, List, Set) and
  - classes (ArrayList, LinkedList, HashSet, TreeSet)
- It also defines an Iterator interface for iterating over these various classes. E.g.

```
LinkedList<Item> list = new LinkedList<Item>() ;
…
Iterator<Item> it = someLinkedList.iterator() ;
while( it.hasNext() ) {
    Item item = it.getNext() ;
    item.doSomthingCool() ; }
```

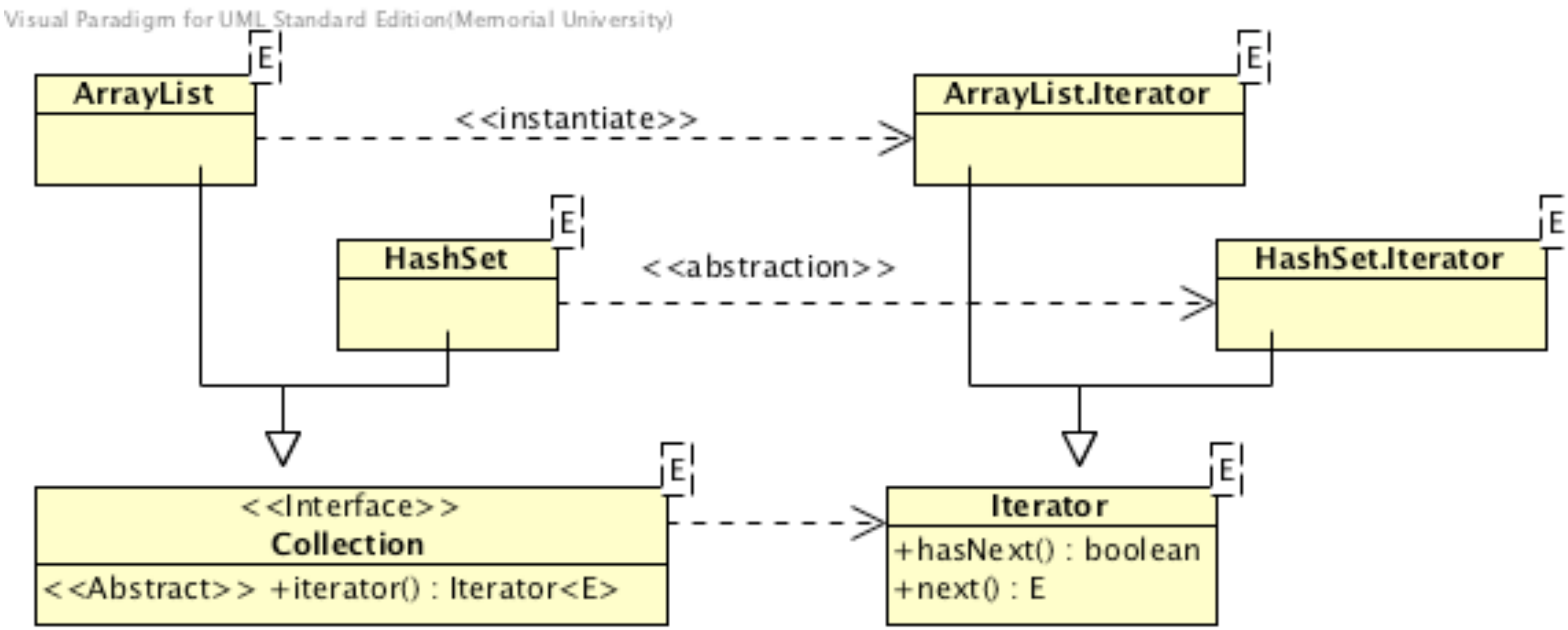
Iterators and Containers (cont.)

- We can also write generic code. E.g.

```
int sum( Collection<Integer> col ) {
    Iterator<Integer> it = col.iterator() ;
    int sum = 0 ;
    while( it.hasNext() ) {
            Integer item = it.getNext() ;
            sum += item.getValue() ; }
    return sum ; }
```

- Each class that implements Collection must implement iterator() to return an iterator object *of the appropriate class*.
- Collection classes are factories for iterators.
- We can add new subclasses of Collection and reuse generic code.

Class diagram for Collections and Iterators

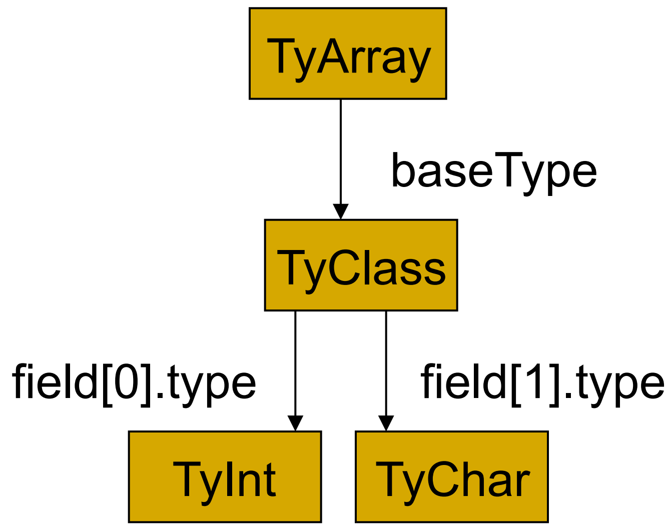


Types and data in the Teaching Machine

- In the TM, C++ types are represented by subclasses of class TypeNode. E.g. TyInt, TyArray, TyClass, etc. These are **composites**.

    struct Pair { int a ; char b ; };

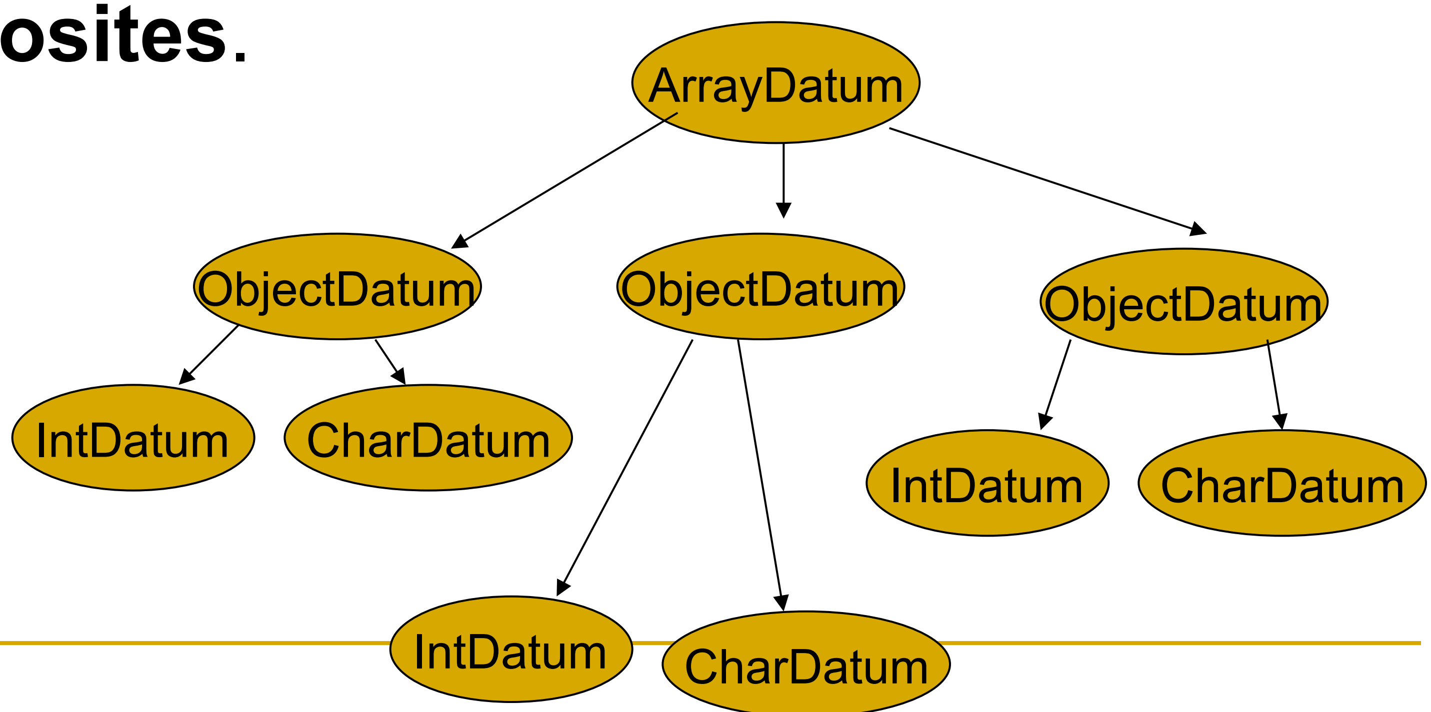
    Pair x[3] ;



Objects representing the type of x

Types and data in the Teaching Machine

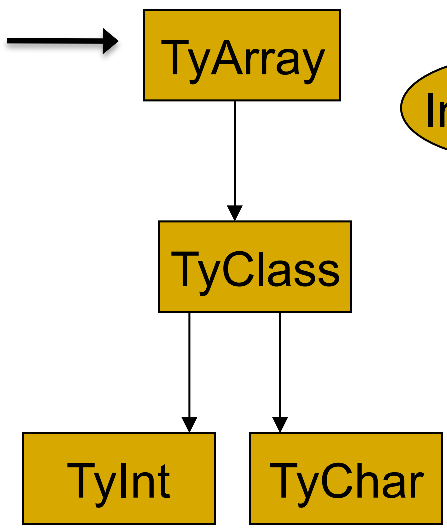
- Variables are represented by subclasses of class AbstractDatum. E.g. IntDatum, ArrayDatum, ObjectDatum. These are also **composites**.



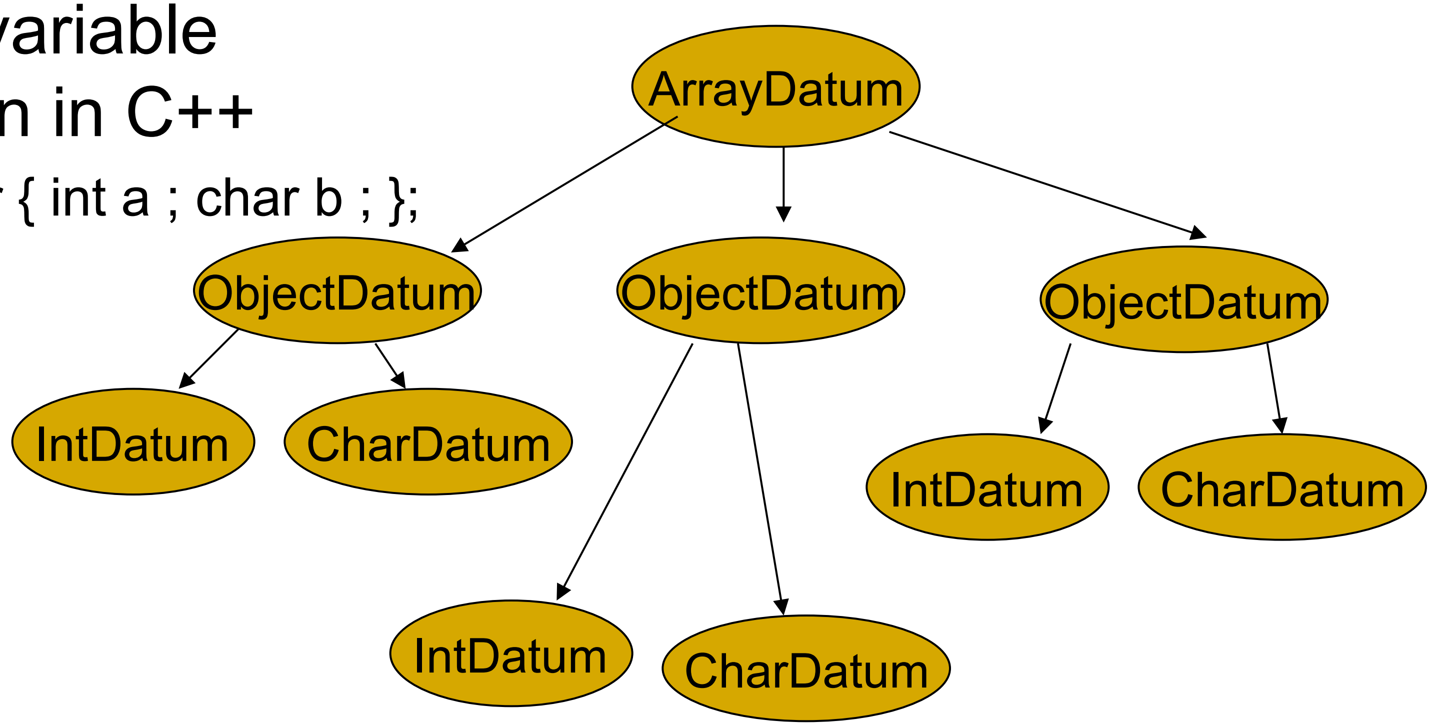
Types and data in the TM (cont.)

Consider a variable declaration in C++

```
struct Pair { int a ; char b ; };
Pair x[3] ;
```

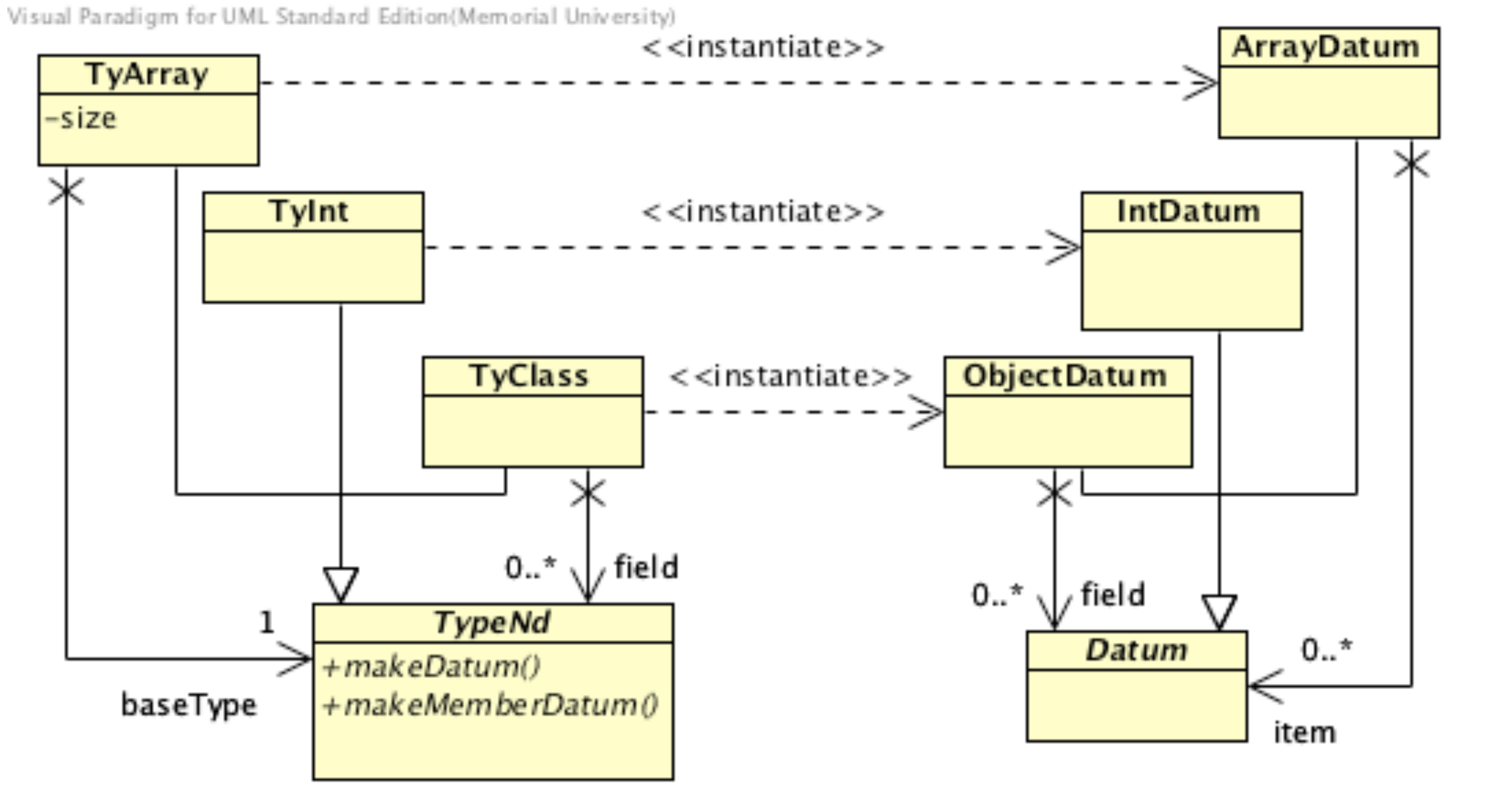


Objects representing the type



When the declaration of x is executed, we need to build objects representing the variable.

Types and data in the Teaching Machine



Types and data in the Teaching Machine

- TypeNodes are **factories** for Datums.
- In TyInt we have

```
Datum makeMemberDatum(Datum parent) { (
    return new IntDatum(parent) ; }
```

- In TyArray, the process is polymorphic with respect to the baseType of the array.

```
Datum makeMemberDatum(Datum parent) {
    ArrayDatum result = new ArrayDatum( parent ) ;
    for( int i = 0 ; i < this.size ; i++ ) {
            Datum item = this.baseType.makeMemberDatum(this) ;
            result.addItem( item ) ; }
    return result ; }
```

Types and data in the Teaching Machine

- In TyClass (used for structs as well as classes), the process is polymorphic with respect to the types of the fields.

```
Datum makeMemberDatum( Datum parent) {
    ObjectDatum result = new ObjectDatum( parent ) ;
    for( int i = 0 ; i < this.numberOfFields ; i++ ) {
            Datum d = this.field[i].type.makeMemberDatum(this) ;
            result.addField(this.field[i].name, d ) ; }
    return result ; }
```

  - Note the polymorphic recursion. This is the *composite pattern* combined with the *factory method pattern*.

Creating Proxies in JSnoopy

- Applications use an Instrumentor object to create proxies from subjects:
- Instrumentor is an abstract class defining a generic factory method

```
abstract class Instrumentor {
   public abstract <T> T instrument( String name,  T subject,
         Class<T> interfaceToInstrument) ; }
```

(Note: For any interface T, Class<T> is the type of the object that represents T at run time and T.**class** points to that object.)

- Clients use an instrumentor to create proxies:

```
this.model = new ConcreteModel() ;
this.model = instrumentor.instrument (
                    "Model", this.model, ModelIntf.class ) ;
```

Creating Proxies in JSnoopy

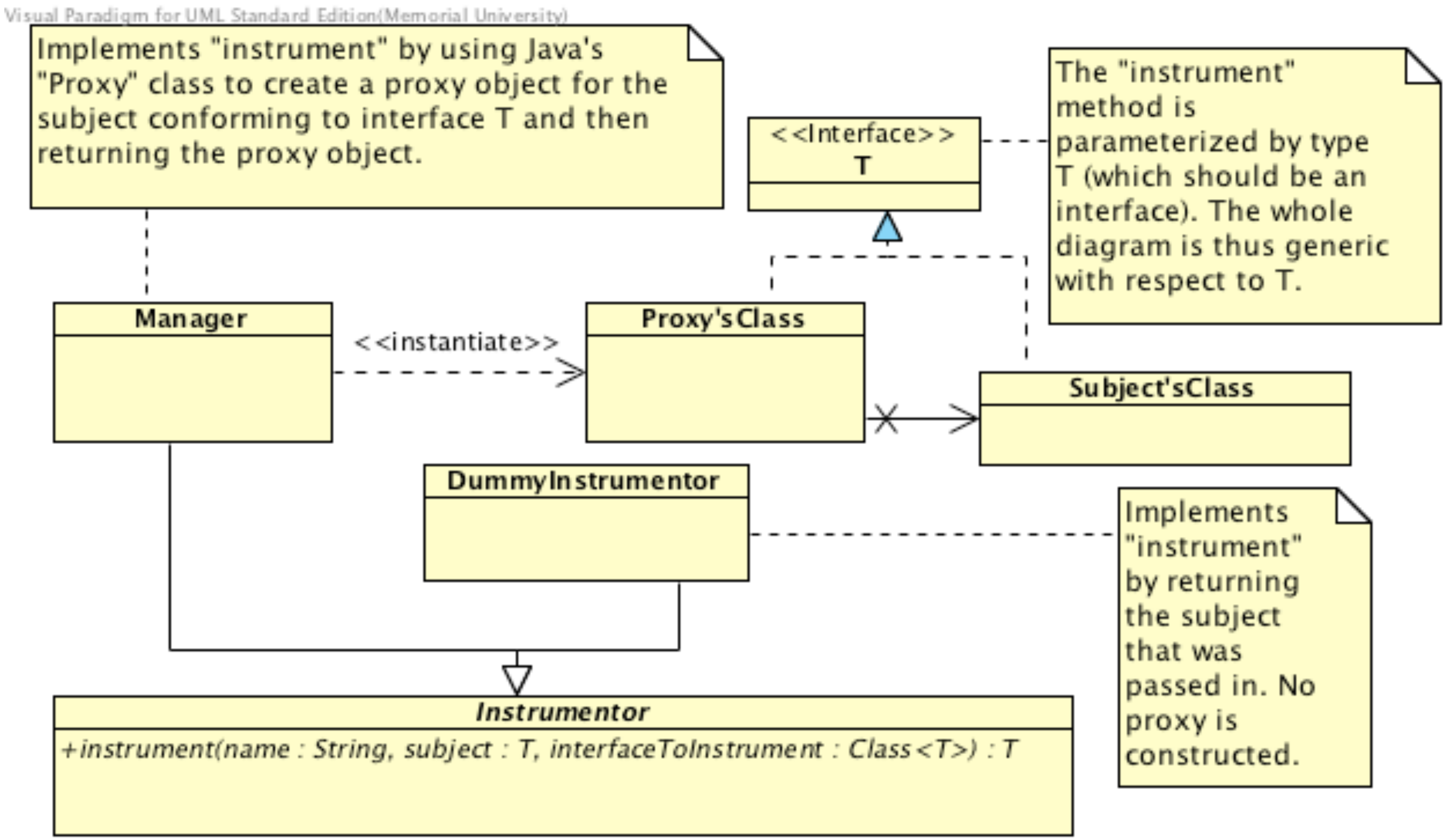
- During regression testing, the application is provided with a instrumentor object of class Manager.

```
class Manager extends Instrumentor {
    @Override
    public <T> T instrument( String nm,  T subject, Class<T> intf)  {
        create and return a proxy for the subject, using Java's very nifty
        java.lang.reflect.Proxy class. } …  }
```

- In the released product, the application is provided with an instrumentor object of class DummyInstrumentor.

```
class DummyInstrumentor extends Instrumentor {
    @Override
    public <T> T instrument( String nm,  T subject, Class<T> intf)  {
        return subject ; } }
```

Creating Proxies in JSnoopy



Factory Pattern Consequences

- (+) Client code can create objects of any of a variety of classes without depending on any of those classes. Hence it is *generic* and *reusable*.
- (+) Connects parallel class hierarchies (e.g. the Collection hierarchy and the Iterator hierarchy; e.g. the TyNode hierarchy and the AbstractDatum hierarchy).
- (+) Whether or not an object is created can be hidden from client. E.g. creator could return a previously created object. (Consider immutable objects)

The Builder Pattern

- Intent: “Separate the construction process for a complex object from its representation so that the same construction process can be used to create different representations.”

Motivating Example

- XML is a common file format for structured documents. However different applications require different internal representations of XML documents.
  - E.g. Xylia is a generic editor for XML documents built on to of the swing.text package. Thus its internal representation extends swing.text.AbstractDocument.
  - But an editor for the Chemical Markup Language (a specific XML language) might require a much different internal representation.
- We would like to use a single parser to read XML files for both these applications.

Motivating Example Solution

- Normally a parser converts a sequence of characters into an object (while checking for syntax errors).
- Instead we write a parser that converts a sequence of characters into a sequence of calls to a pluggable object.

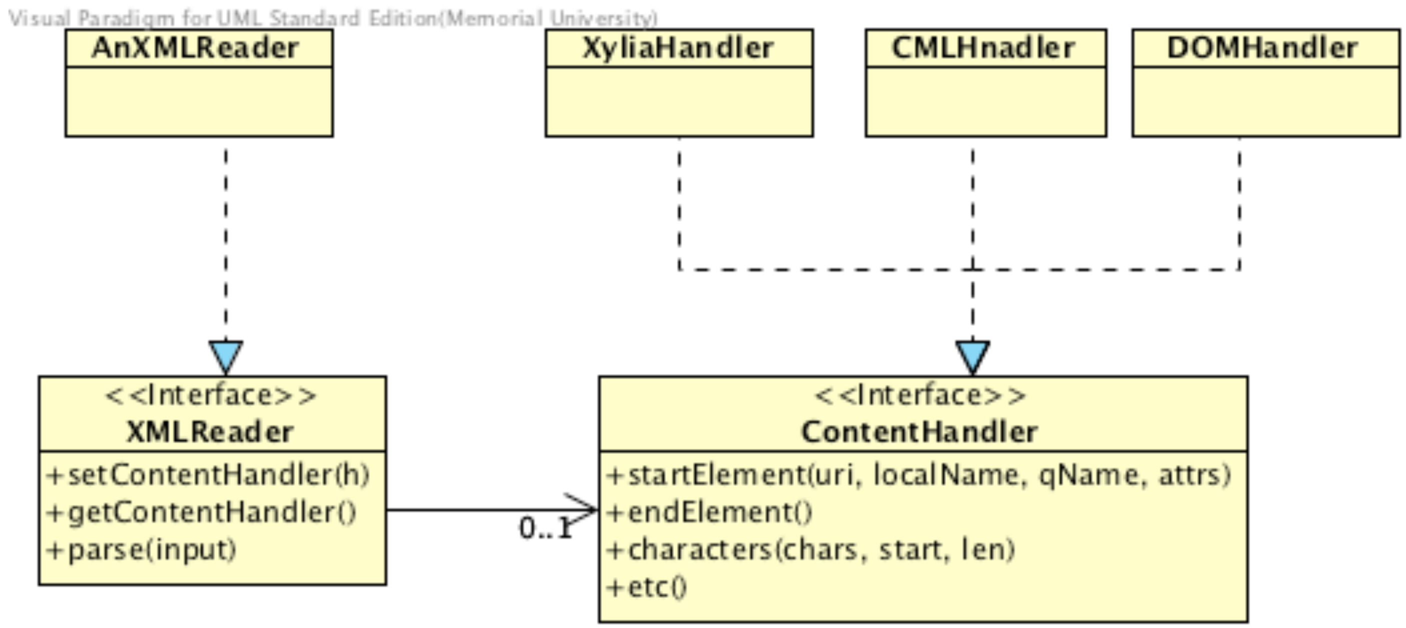
Solution (cont.)

- <xhtml><body><p>hi</p></body></xhtml>
- Is converted to calls

```
handler.startDocument() ;
handler.startElement(…,“xhtml”,…) ;
handler.startElement(…,“body”,…) ;
handler.startElement(…,“p”,…) ;
handler.characters(“hi”) ;
handler.endElement() ; handler.endElement() ;
handler.endElement; handler.endDocument() ;
```

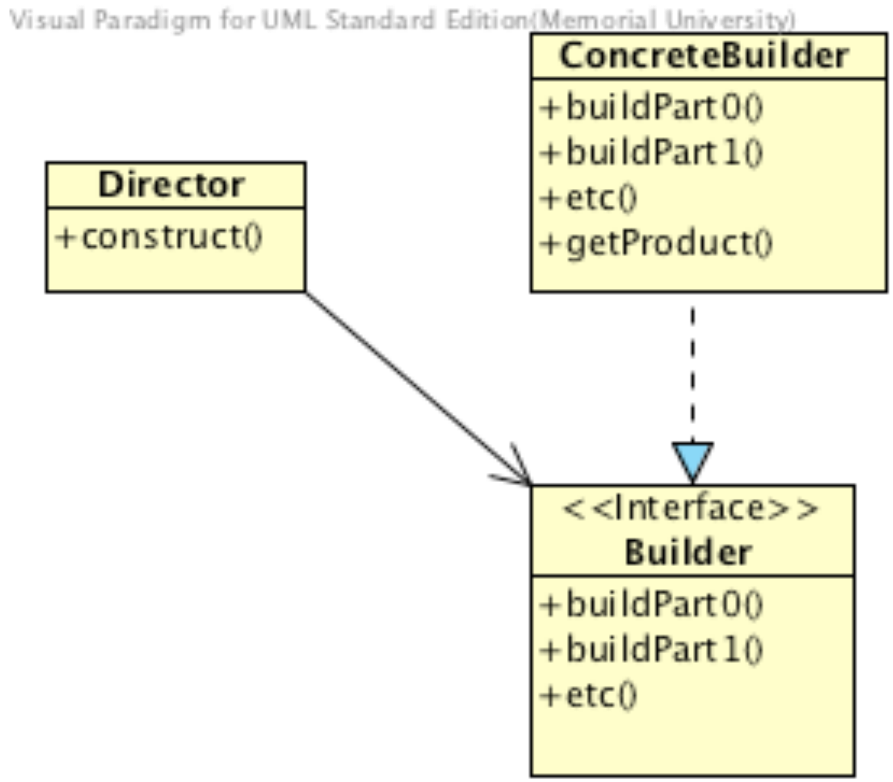
- Here “handler” is the pluggable object

The UML Class diagram



The general pattern

- The director sends a series of commands to its Builder object instructing how to build a product.
- Any object that realizes the Builder interface can be used to construct the object.



SAX

- SAX (Simple API for XML) uses the Builder pattern to specify a common standard for XML parsers
  - A number of different parsers all use SAX.
  - A large number of XML based applications use SAX.
- SAX allows multiple handler (builder) objects. Allows different handlers to pay attention to different commands.

Example: Content Model Parser

- Content models in XML are regular expressions describing the sequence of children an element can have. E.g. (p | table | img)* is a (simplified) content model for an xhtml “body”.
- Content models are converted to a sequence of calls to a pluggable builder object via an interface. The interface is optimized for building trees:

Content Model Parser (the builder interface.)

The parser calls these methods after parsing the corresponding construct.

Object finishContentModel(int kind) ;

Object finishContentModel(int kind, Object regExp) ;

Object mkPCDATA( ) ;

Object mkName(String name) ;

Object mkAlternation(Object left, Object right) ;

Object mkSequence(Object left, Object right) ;

Object mkOptional(Object operand) ;

Object mkKleeneStar(Object operand) ;

Object mkKleenePlus(Object operand) ;

Content Model Parser (cont.)

Parsing ((a,b) | c)*

The default builder (factory) simply builds a tree, as shown at right.