Polymorphism (Chapter 6)

Prof. Dr. Wolfgang Pree

Department of Computer Science
University of Salzburg
cs.uni-salzburg.at

Polymorphism (1)

- Interface - requires no more than is essential for the service to be provided.

- Clients and providers - free to overfullfill their contract.
- Client
  - may establish more than is required by the precondition
  - may expect less than is guaranteed by the postcondition.
- Provider
  - may require less than is guaranteed by the precondition
  - may establish more than is required by the postcondition.

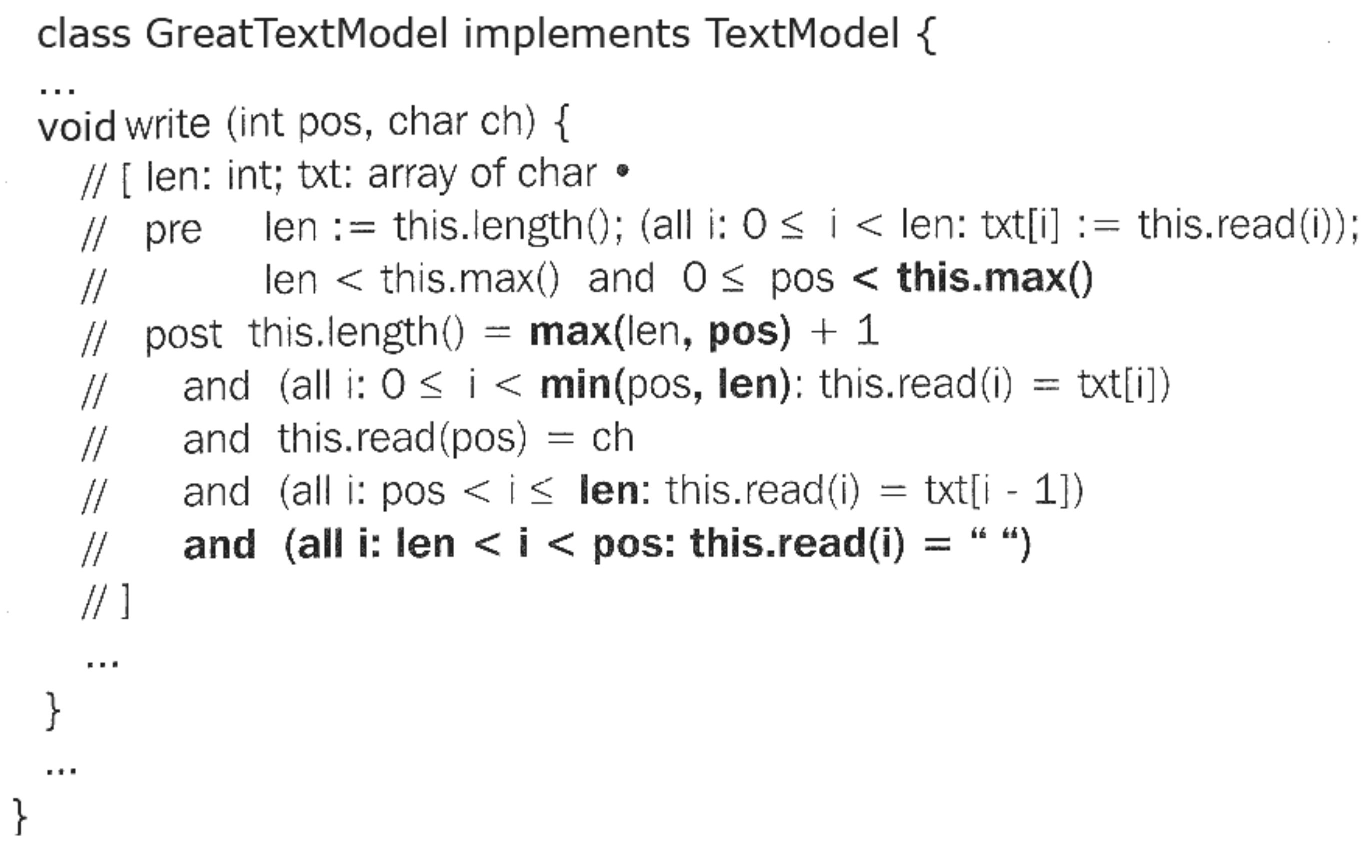
- Pre- and postconditions are usually specified using predicates.

Polymorphism (2)

```
interface TextModel {
  int max ();   // maximum length this text can have
  int length ();   // current length
  char read (int pos);   // character at position pos
  void write (int pos, char ch);   // insert character ch at position pos
    // [ len: int; txt: array of char •
    //   pre    len := this.length(); (all i: 0 ≤ i < len: txt[i] := this.read(i)) :
    //          len < this.max() and 0 ≤ pos ≤ len
    //   post  this.length() = len + 1
    //     and  (all i: 0 ≤ i < pos: this.read(i) = txt[i])
    //     and  this.read(pos) = ch
    //     and  (all i: pos < i < this.length(): this.read(i) = txt[i - 1])
    // ]
  ...
}
```

- The interface requires callers of operation *write* to make sure that the character to be written is inserted within the current range of the text.

Polymorphism (2 – example)



```
class GreatTextModel implements TextModel {
...
void write (int pos, char ch) {
   // [ len: int; txt: array of char •
   //   pre    len := this.length(); (all i: 0 ≤ i < len: txt[i] := this.read(i));
   //          len < this.max()  and  0 ≤ pos < this.max()
   //   post  this.length() = max(len, pos) + 1
   //      and  (all i: 0 ≤ i < min(pos, len): this.read(i) = txt[i])
   //      and  this.read(pos) = ch
   //      and  (all i: pos < i ≤ len: this.read(i) = txt[i - 1])
   //      and  (all i: len < i < pos: this.read(i) = " ")
   // ]
   ...
 }
 ...
}
```

- An implementation *GreatTextModel* may relax this by allowing insertions to happen past the end of the current text, by padding with blanks where necessary.

Polymorphism (3)

```
    ( from precondition:
        len := this.length(); max := this.max();
        (all i: 0 ≤ i < len: char[i] := this.read(i))
    )
    len < max  and  0 ≤ pos ≤ max
  and
    this.length() = max(len, pos) + 1
    and  (all i: 0 ≤ i < min(pos, len): this.read(i) = char[i])
    and  this.read(pos) = ch
    and  (all i: pos < i ≤ len: this.read(i) = char[i - 1])
    and  (all i: len < i < pos: this.read(i) = " ")
  ⇒
  this.length() = len + 1
  and  (all i: 0 ≤ i < pos: this.read(i) = char[i])
  and  this.read(pos) = ch
  and  (all i: pos < i < this.length(): this.read(i) = char[i - 1])
```

- A provider may accept arbitrary positions, including negative ones, in all operations, simply by first clipping the specified position to the currently valid range.
- Alternatively, a provider may decide to grow a text dynamically, making *max* return a non-constant value.

- Equally, a provider may decide to create the illusion of an infinitely long text preinitialized with all blanks.
- There are, surprisingly, many further possible ways in which providers can interpret the interface contract.

Polymorphism (4)

```
{ TextModel text; int pos; char ch;
  ...
  if text.length() < text.max() {
    text.write(text.length(), ch);
    // expect  text.read(text.length() - 1) = ch
  }
  ...
}
```

- On the client's end of an interface, the same sort of relaxation is possible. A client may guarantee more than is required and expect less than is provided.
- A client of *TextModel.write* may use the operation only to append to a text.

Polymorphism (5)

- The flexibility introduced by requiring only implications, instead of equivalences, becomes very important when considering interactions of multiple providers and clients.
- Libraries always supported the concept of a service provider catering for many clients.
- In any one given configuration, there is only one implementation of a library interface and the only concern on the provider’s side is versioning.
- The situation has changed with the introduction of self-standing interfaces and dynamic dispatch (late binding).

UNIVERSITÄT SALZBURG

Types, subtypes, and type checking (1)

- Ideally
  - all conditions of a contract would be stated explicitly and formally as part of an inerface specificaion.
  - Highly desirable to have a compiler or other automatic tools check clients and providers against the contract and reject incorrect ones.

- Fully formalizing the interface contracts is a first major obstacle.
- Automatic checking remains a major challenge of ongoing research. An efficient general-purpose verifier is not feasible.
- Theorem proving
  - Too expensive to be done in a production compiler.
  - Requires manual assistance by an expert.

Types, subtypes, and type checking (2)

- For example, version conflicts among independently provided components can only be checked at configuration or load time. Likewise, index range errors can, generally, only be detected at runtime.
- By combining a type system with automatic memory management and certain runtime checks, a language implementation can fully eliminate memory errors – and some other error classes as well. Smalltalk, Java, C#, Oberon fall into this category. However many others, Modula-2, C and C++, do not.
- In the context of objects and interfaces, a type is best understood as a set of objects implementing a certain interface.
- The parameters form a part of an operations's preconditions. The types of output and in-out parameters plus the type of returned values form a part of an operation's postconditions.
- Thus, a type is an intervace with a simplified contract.

Types, subtypes, and type checking (3)

```
interface View {
   void close ();
   void restore (int left, int top, int right, int bottom);
}
interface TextView extends View {
   int caretPos ();
   void setCaretPos (int pos);
}
interface GraphicsView extends View {
   int cursorX ();
   int cursorY ();
   void setCursorXY (int x, int y);
}
```

- An object may implement more than the interface required in a certain context. It may implement additional interfaces or a version of the required interface extended with additional operations or both.

Types, subtypes, and type checking (4)

- A client expecting a certain type really expects an object fulfilling a contract. Thus, all objects fulfilling the contract could be used. On the level of types, this includes all extended types as long as the understanding is that objects of an extended type respect the base contract.
- Interface inheritance is the most common way to form subtypes. Another way is structural subtyping, in which no base interface is named. Instead, operations are repeated and a subtype is said to be formed if a subset of the operations coincides with operations defined for another type *View, TextView, GraphicsView*, or other.

  This is called polymorphism, and the variable is said to be of a polymorphic type. As there are several forms of polymorphism, this particular one is also **called subtype or inclusion polymorphism**.

Contra :: Co Variance (1)

- http://c2.com/cgi/wiki?ContraVsCoVariance:
- To summarize the concept: When inheriting from a class and over-riding one of its methods, how should you constrain the types accepted as arguments and returned as results according to the method being over-ridden?
- Say you have a class Foo, which has a method bar(). Method bar() takes an argument of type middle_type, and returns a value of type middle_type. Now you make a subclass of Foo called SubFoo, and you override bar(). What types can the new bar() take? What types can it return?
- Look at return types first: we want to be able to substitute SubFoo where existing code expects Foo, so it needs to return things of type middle_type, or of some subtype (e.g. sub_type). This should be pretty obvious.

Contra :: Co Variance (2)

- As for what types our new bar() can take: One answer (cf. C++) is: bar() can take only things of type middle_type. You can't declare it to take sub_type, and you can't declare it to take super_type. This is called ***invariance***.
- Another answer: bar() can only be declared to take things that are a *subtype* of middle_type - so middle_type is OK, and sub_type is OK, but super_type is out. This is called ***covariance***.
- Finally, the third answer: bar() can only be declared to take things that are a *supertype* of middle_type - so any of middle_type, super_type, and sub_type may be passed to bar(). This is called ***contravariance***.

Contra :: Co Variance (3)

- Covariance seems to jibe with our notion that subclasses are more specialized, less general than their superclasses. So you might have a Collection class which takes and returns Objects; you could subclass it to make a FooCollection class which takes and returns Foos (where Foo is a subclass of Object).
- Contravariance sounds kind of counterintuitive at first, but it's actually just analogous to the famous advice about implementing protocols: "Be liberal in what you accept, and conservative in what you send." So just as you have to *return* a subtype of the original bar()'s return type, you have to *accept* any supertype of whatever the original bar() accepts. If your type system enforces contravariance of parameters, then it can tell at compile time whether your code is typesafe (cf. SatherLanguage, ObjectiveCaml, CeePlusPlus (since invariance is just a special case of contravariance)). If it enforces covariance, it can't really do that, but it can make some good guesses (cf. EiffelLanguage) - though Eiffel is known to crash when it guesses wrong.

Covariance and contravariance : conflict without a cause Castagna, Giuseppe ACM Transactions on Programming Languages and Systems Vol.17, No. 3 (May 1995), pp. 431-447

More on subtypes (1)

- A provider may establish more than is required by a contract. Hence, a subtype interface can replace the types of output parameters and return values by something more specific.
- As the types of output parameters and return values can thus be varied in the same direction as the types of the containing interfaces, this is called covariance.
- A provider may expect less than is guaranteed by a contract. Hence, a subtype interface could replace the types of input parameters by something more general - supertypes. In other words, the types of input parameters may be varied from types to supertypes when going from an interface of a certain type to an interface of a subtype of that type. As the types of input parameters can thus be varied in the opposite direction of the types of the containing interfaces, this is called contravariance.

More on subtypes (2)

- Types of in-out parameters simultaneaously from part of an operation's pre- and postcondition.
- It follows that types of in-out parameters cannot be varied at all in a subtype interface, sometimes called invariance of in-out parameter types.
- Covariantly redefining *getModel* in *TextView*. The same can be done for *getModel* in *GraphicsView*.
- This is obviously useful. Clients that care only about *View* will get a generic *Model* when they ask for the views model. However, clients that know they are dealing with a *TextView* object will get a *TextModel*.

```
interface View {
  ... // as above
  Model getModel ();
}
interface TextView extends View {
  ... // as above
  TextModel getModel ();
}
interface GraphicsView extends View {
  ... // as above
  GraphicsModel getModel ();
}
```

More on subtypes (3)

```
interface View {
 … // as above
 void setModel (Model m);  // is this a good idea?
}
```

- However, a *TextView* object will get a *TextModel*. If covariant change of input parameter was safe, *setModel* could be changed in *TextView* to accept a *TextModel*.

Object languages and types

- Some languages, such as Smalltalk, do not have an explicit type system.
- A compiler can still derive types by inspection of strictly local program fragments, such as in the Smalltalk dialect StrongTalk. In such languages, avoidance of explicit typing is a matter of convenience – ther is less to write.
- Adding explicit typing is still useful, as it makes important architectural and design decisions explicit.
- Modern languages, such as Java use an explicit type system and statically check programs at compile time. In addition, they check narrowing type casts at runtime.

- Few of the mainstream languages support any changes in types of operations when forming subtypes. In C++, covariant return types were introduced only in early 1994. Java and C# still do not support any type changes.

UNIVERSITÄT SALZBURG

The paradigm of independent extensibility (1)

- The principle function of component orientations is to support independent extensibility.
- **A system is independently extensible if it is extensible and if independently developed extensions can be combined.**
  For example, all operating systems are one-level independently extensible by loading applications. Extensions to applications are sometimes called plugins.
- Combining the “dekernelization” efforts of operating system architects and the modularization efforts of application architects leads to a new vision for overall system architectures – components everywhere! It is all about forming a system architechture that is independently extensible on all levels.

The paradigm of independent extensibility (2)

- Partitioning of systems into smallest components (to maximize reuse) conflicts with efficiency, but also with robustness when facing evolution and configurational variety.
- Micro-kernel architecture enforces total isolation of application-level processes to establish system safety and support security mechanisms. Frequent crossing of protection domain boundaries can severely affect performance.
- Overall system performance directly conflicts with extreme micro-kernel designs.

The paradigm of independent extensibility (3)

- How can component technology and independent extensibility as a recursive system design concept ever be viable if performance is so severely affected?
- It turns out to be the wrong question.
- True question: “Why is performance so severely affected?”
- Obvious answer: “because it is expensive to perform cross-context calls.”
- A cross-context call on well-tuned operating systems is still easily a hundred times more expensive than local in-process call.

The paradigm of independent extensibility (4)

- Personal computers: Neither the Mac OS nor MS-DOS had a true process model. Hardware protection was mostly ignored. A malicious program could easily “crash” the entire system.
- Considered acceptable as these machines typically serve a single user who can avoid the crash simply by avoiding the use of unreliable applications.
- *It is possible to have your cake and eat it too?* Could there be a third way?
  One way is to choose carefully the granularity of components. Another way is to guarantee statically that a component will be safe.

UNIVERSITÄT SALZBURG

Safety by construction – viability of components (1)

- The discussion on type safety points in the right direction.
- Here is an example. The Java class files (Java's portable compiled format) and the Java virtual machine have been crafted to interact in a way that prevents type-unsafe applets from being executed in a non-local environment.
  - Java is a type-safe language.
  - It provides automatic memory management using garbage collection.
  - Finally, it performs runtime checks on all operations that are "dangerous", but cannot be statically checked, such as array bounds checks.
- Together, these techniques guarantee that memory errors cannot occur. As class files, produced by the Java compiler, could be tampered with the virtual machine rechecks them when loading one coming from a non-local site, such as across the internet.

Safety by construction – viability of components (2)

- Another example is the Microsoft .NET Common Language Runtime (CLR). CLR also uses late compilation and avoids interpretation, eliminating the notion of a virtual machine. It defines an intermediate language that has been designed to support a wide variety of programming languages and still map efficiently to a wide variety of processors.
- Strong safety properties can be established at compile-time, checked at install, load, or JIT compile-time, and then be relied on without further checking while executing the resulting efficient code.

Module safety

- Type safety and elimination of memory errors are not enough, it would still be possible for a program to call arbitrary services present in or loadable into the system.
- One additional requirement: module safety. A component has to specify explicitly which services – from the system or other components – it needs to access. This is done in the form of module (or package) import lists. The language and system do not allow access to any non-imported module. This is like access control in file systems.
- Module safety is **not quite as simple** as it sounds. In component systems it is important that other components (and services) can be retrieved by name. A clean and popular way to support components retrieval by name is a **reflection service**. Java, Component Pascal, and .NET CLR all offer such a service. Where access to components is to be restricted, **reflection services require special attention so as not to create a security loophole.**

Module safety and meta-programming

- Where meta-programming interfaces exist, these need to be explicitly restricted such that these services do not break encapsulation. This is contrary to some of the typical usages of meta-programming, such as debugging or data structure serialization services.
- **A system may offer two meta-programming interfaces**
  - one that is **module safe** and open for general use
  - and another that is **module unsafe and restricted to trusted components.**

Safety, security, trust (1)

- Type safety, module safety, and absence of memory errors – trustworthy?
- If the target is a set of components, installed locally, interacting on a personal computer, then this approach may already be close to satisfactory. If the target was the highest security level, then this approach would be totally unacceptable.
- Low- to medium-security system – this approach is probably at its limits. What is wrong? First, and above all, this approach fully relies on the tight semantics of the language. To be fully trustworthy a formal semantics of the language together with formal proofs of the claimed safety properties would be required. Very few programming languages satisfy this requirement and there is no mainstream object-oriented language.

Safety, security, trust (2)

- A language implementation could still break any proved property. To go all the way, the language processors and the language runtime systems again need to be formally specified and verified.
- Trust is a matter of reducing the unknown to the known and trusted, and doing so in a trusted way. This is obviously primary a sociological process.
- Unix security mechanism: The designers of the mechanism published its details in full and encouraged everyone to try to break it. After years, the mechanism gained (or, better, earned) the trust it currently receives.
- The Java designers also publicized their security strategies early on and encouraged serious research groups to challenge the approach. After years of steadily decreasing reports of found problems, people will increasingly trust the approach.

Bottleneck interfaces

- Interfaces introduced to allow the interoperation between independenty extended abstractions are somethimes called bottleneck interfaces (cf Adapter design pattern).
- A bottleneck interface is a self-standing contract. As a bottleneck interface couples independent extension families, it cannot itself be extended.
- Once published, it can only be withdrawn or replaced but not extended.

Evolution versus immutability of interfaces and contracts

- As soon as a contract has been published to the world, it (the interface and its specification) can no longer be changed. This holds for clients and providers bound by a specific contract.
- A provider can always stop providing a particular interface. It will then potentially lose part of its client base – the part has not yet been migrated to some newer interface. However, a provider can never change the specification of an existing contract as that would break clients without any obvious indication.

Syntactic versus semantic contract changes (1)

- Changes to a contract can take two forms. Either the interface of the specification is changed. **If the interface is changed, this is referred to as a syntactic change.**
- **If the specification is changed, this is called a semantic change.**
- Typical providers in object-oriented settings are classes, the problem caused by contract change is sometimes referred to as the **fragile base class problem**.
- A simple way to avoid these problems is to refrain from changing contracts once they have been published.

UNIVERSITÄT SALZBURG

Syntactic versus semantic contract changes (2)

- It is again helpful to consider the analogy of traditional contracts. No clause in a contract, once signed, can be changed without the agreement of all involved parties. Of course, once there is an uncontrollable number of parties, gaining agreement can become difficult or impossible. However, such contracts do have mechanisms that allow for change. The two fundamental mechanisms are
  - acknowledging existence of overriding law and instances, and
  - statement of a termination time.
- Today, only Microsoft's COM declares all published interfaces to be immutable. Eventually such older interfaces can and should go. IBM's SOM did something different. A new release can only add to an interface, it cannot take functionality away.

Contract expiry

- Some of the current component infrastructures offer licensing services and it is a natural property of licenses that they expire after a preset date.
- Istead of supporting legacy contracts forever, adding more baggage to providers and clients, there is a clean way to cut off the past. However, without mutually agreed expiry dates, this will always come as a surprise to some users.

Other forms of polymorphism

- **Overloading** groups otherwise unrelated operations (C++)
- Using the same implementation to serve a variety of types: A list implementation can be parameterized with the type of the list elements. The list implementation itself is generic and provided only once. This is called **parametric polymorphism**.
- Parametric polymorphism is similar to C++ templates. However, C++ templates lead to code explosion as a template is necessarily compiled to different code for each instantiation. Also, templates cannot be statically type-checked as type checking cannot occur before parameters are supplied.