Rigorous Software Engineering

Modeling and Specifications

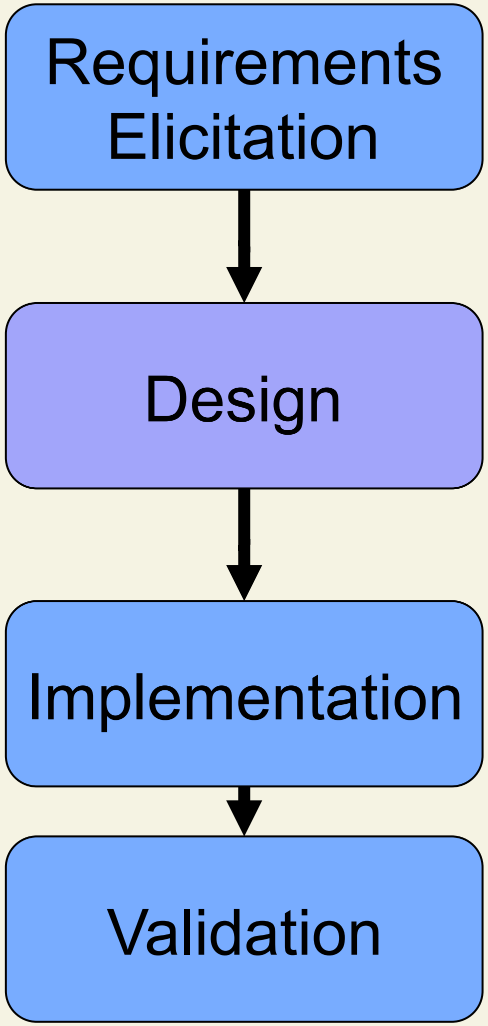
Prof. Zhendong Su

(based on slides from Prof. Peter Müller)

Spring 2019

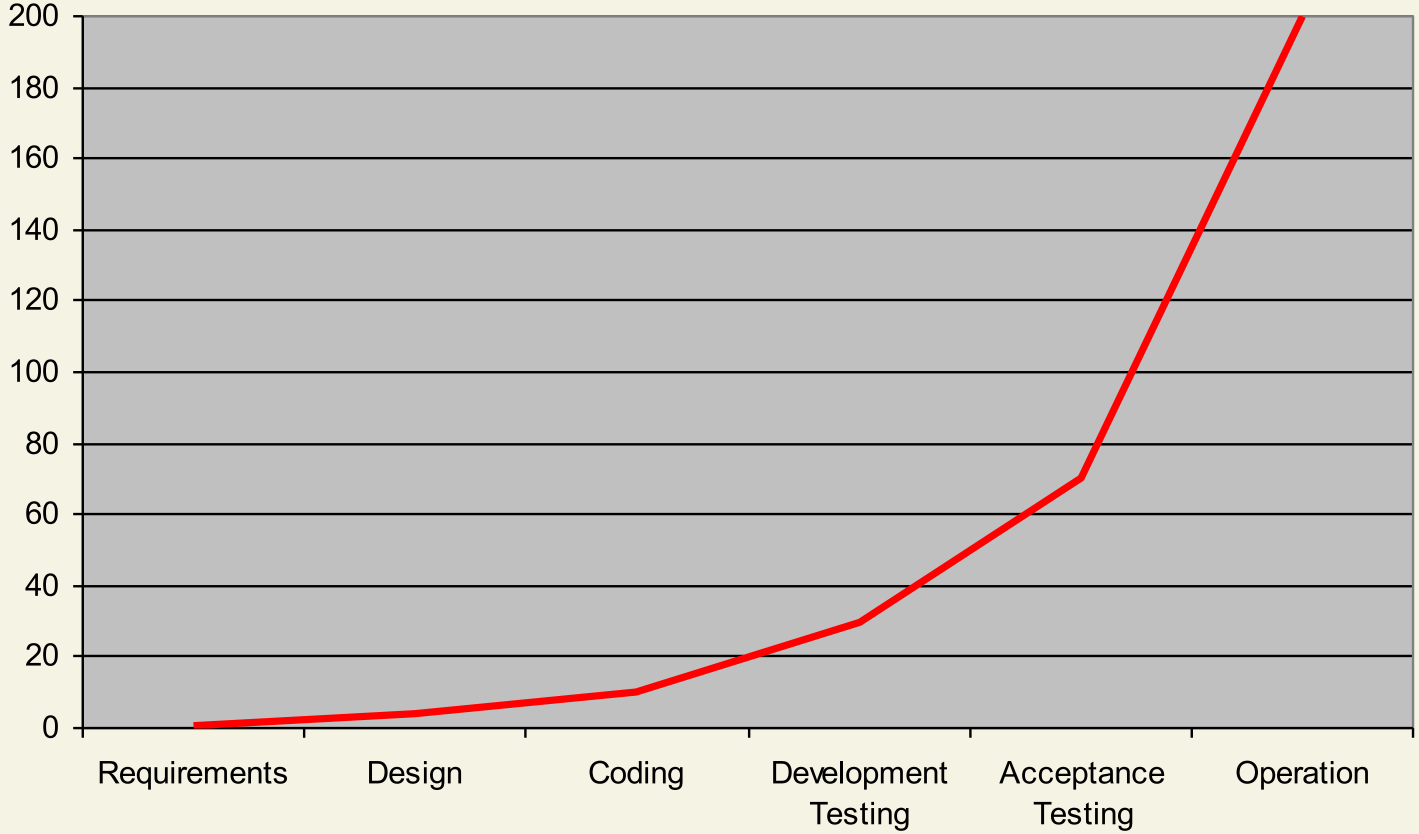
ETH zürich

Main Activities of Software Development



Relative Cost to Fix an Error

- The sooner a defect is found, the cheaper to fix it



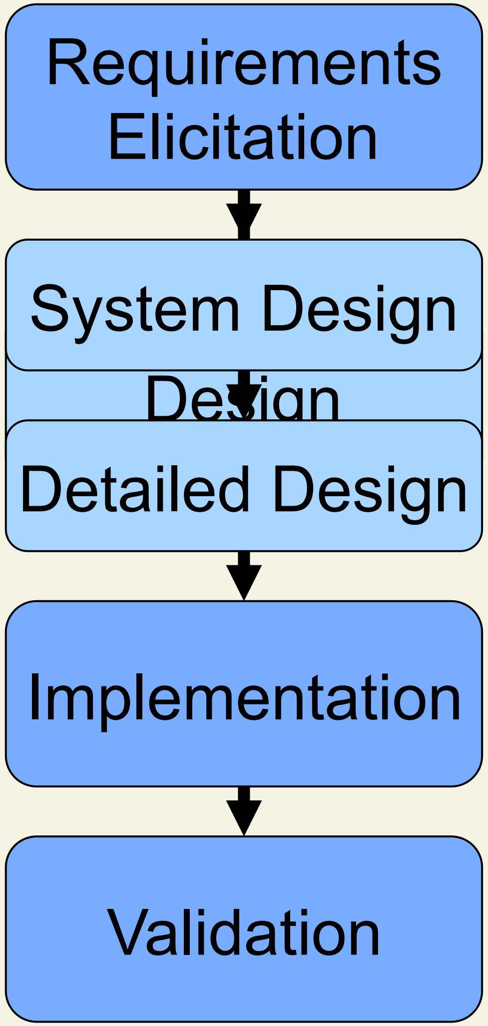
[Boehm 1981]

Mastering Complexity

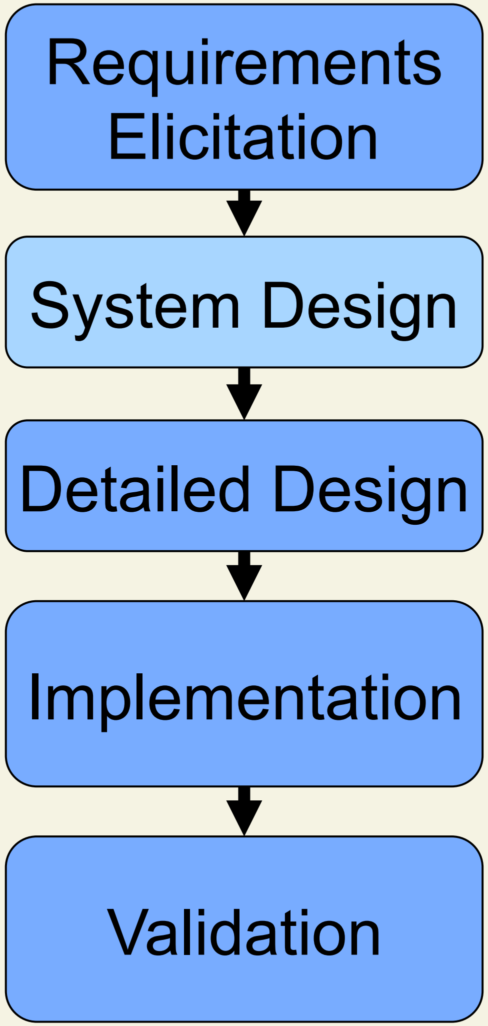
- *The technique of mastering complexity has been known since ancient times: Divide et impera (Divide and Rule)* [Dijkstra 1965]

- Benefits of decomposition
  - Partition the overall development effort
  - Support independent testing and analysis
  - Decouple parts of a system so that changes to one part do not affect the other parts
  - Permit system to be understood as a composition of mind-sized chunks with one issue at a time
  - Enable reuse of components

Main Activities of Software Development



System Design

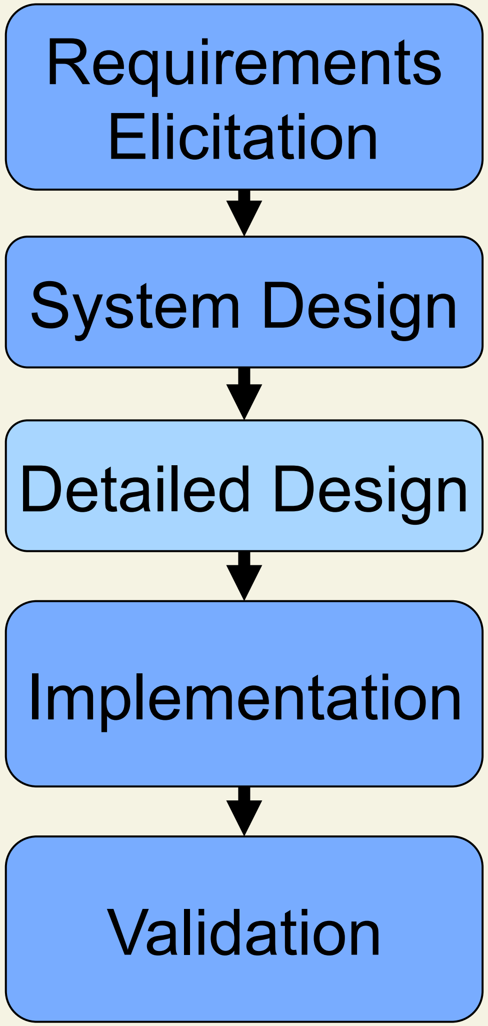


- System design determines the software architecture as a composition of sub-systems

- Components: Computational units with specified interface
  - Filters, databases, layers

- Connectors: Interactions between components
  - Method calls, pipes, events

Detailed Design



- Detailed design
  - Chooses among different ways to implement the system design
  - Provides the basis for the implementation
- Data structures
- Algorithms
- Subclass hierarchies

Detailed Design: Map Example

- Is **null** permitted as a value in the hash map?
- Is it possible to iterate over the map?
  - Is the order of elements stable?
- Is the implementation thread-safe?

```
package java.util;
class HashMap<K,V> … {
  V get( Object key ) { … }
  V put( K key, V value ) { … }
  …
}
```

```
HashMap<String, String> m =
   new HashMap<String, String>( );
m.put( "key", null );
String r1 = m.get( "key" );
String r2 = m.get( "no key" );
```

Map Example: Some Design Alternatives

- #1: permit **null**-values
  If key is not present, get
  - returns **null** (Java)
  - throws an exception (.NET)
  - indicates this via a 2nd result value (e.g., an "out" parameter in C#)

```
HashMap<String, String> m =
   new HashMap<String, String>( );
m.put( "key", null );
String r1 = m.get( "key" );
String r2 = m.get( "no key" );
```

- #2: do not permit **null**-values
  If **null**-value is passed, put
  - throws an exception
  - does nothing

Detailed Design: Initialization Example

- Initialize fields of an object
  - When the object is created, or
  - When the fields are accessed for the first time?

```
class ImageFile {
  String file;
  Image image;

  ImageFile( String f ) {
    file = f;
    // load the image
  }

  Image getImage( ) {
    return image;
  }
}
```

```
class ImageFile {
  String file;
  Image image;

  ImageFile( String f ) {
    file = f;
  }

  Image getImage( ) {
    if( image == null ) {
      // load the image
    }
    return image;
  }
}
```

Detailed Design: List Example

- Do operations mutate the data structure?

```
void demo( List<String> l ) {
  l.set( 0, "Hello " );
  foo( l.take( ) );
  String s = l.get( 0 ).trim( );
}
```

May foo execute concurrently?

What is the runtime and memory overhead?

May foo modify l?

List Example: Destructive Updates

```
class List<E> {
  E[ ] elems;
  int len;

  List( E[ ] e, int l ) {
    elems = e; len = l;
  }

  void set( int index, E e )
  { elems[ index ] = e; }

  List<E> take( ) {
    return new List<E>( elems, r.len – 1 );
  }
}
```

Destructive Updates in Action

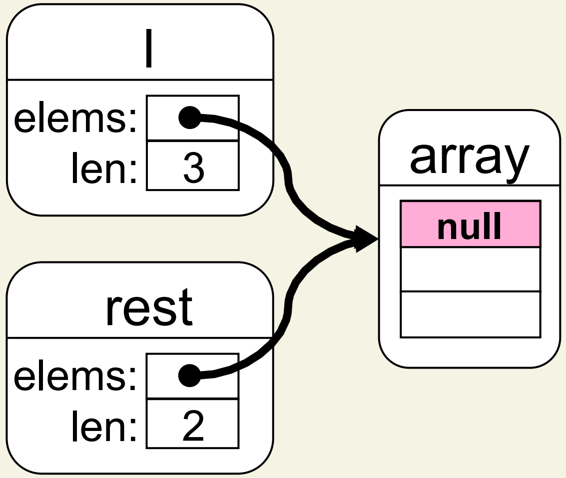
Concurrency may lead to data races

take requires constant time and space

```
void demo( List<String> l ) {
  l.set( 0, "Hello" );
  foo( l.take( ) );
  String s = l.get( 0 ).trim( );
}
```

Side effects become visible

```
void foo( List<String> p ) {
  p.set( 0, null );
}
```



List Example: Copy-on-Write

```
class List<E> {
  E[ ] elems;
  int len;

  List( E[ ] e, int l ) {
    elems = e; len = l;
  }

  void set( int index, E e ) {
    elems = elems.clone( );
    elems[ index ] = e;
  }

  List<E> take( ) {
    return new List<E>( elems, r.len – 1 );
  }
}
```

Copy-on-Write in Action

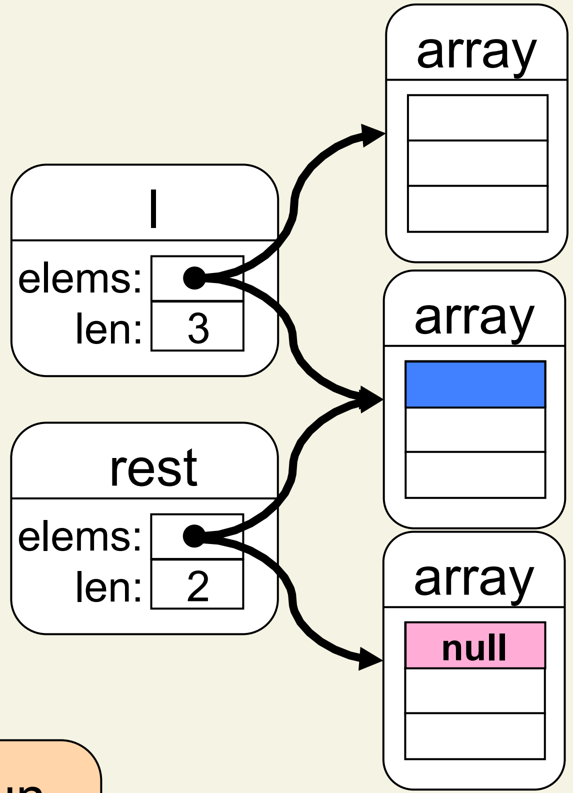
Concurrency is safe

```
void demo( List<String> l ) {
  l.set( 0, "Hello" );
  foo( l.take( ) );
  String s = l.get( 0 ).trim( );
}
```

No side effects on l

```
void foo( List<String> p ) {
  p.set( 0, null );
}
```

Significant run-time and space overhead



List Example: Reference Counting

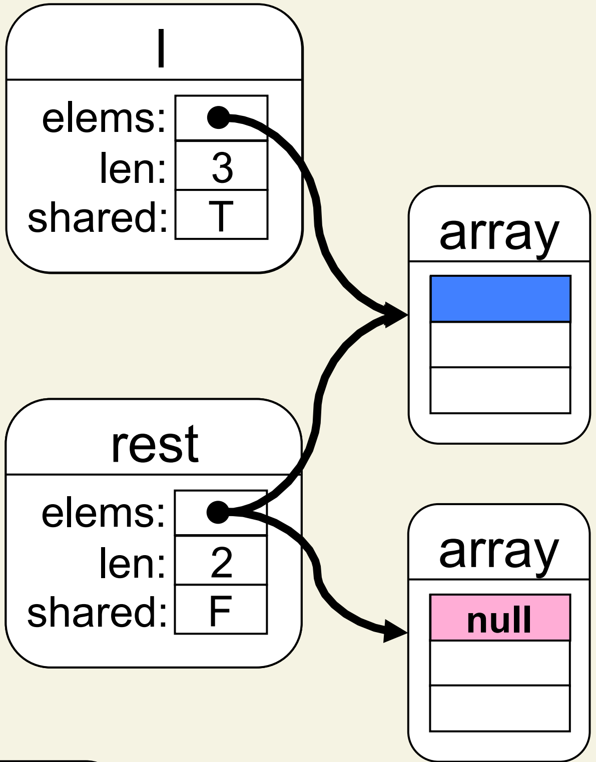
```
class List<E> {
  E[ ] elems; int len;
  boolean shared;

  List( E[ ] e, int l ) {
    elems = e; len = l; shared = true;
  }

  void set( int index, E e ) {
    if( shared )
    { elems = elems.clone( ); shared = false; }
    elems[ index ] = e;
  }

  List<E> take( ) {
    shared = true;
    return new List<E>( elems, r.len – 1 );
  }
}
```

Reference Counting in Action

Concurrency is in general unsafe

```
  id demo( List<String> l ) {
  set( 0, "Hello" );
  foo( l.take( ) );
  String s = l.get( 0 ).trim( );
}
```

No side effects on l




```
void foo( List<String> p ) {
  p.set( 0, null );
}
```

Less run-time and space overhead

ETH zürich

3. Modeling and Specification

3.1 Code Documentation

3.2 Informal Models

3.3 Formal Models

Design Documentation

- Design decisions determine how the code should be written
  - During initial development
  - When extending code through inheritance
  - When writing client code
  - During code maintenance

- Design decisions must be communicated among many different developers
  - Does source code convey design decisions appropriately?

Example: Using HashMap

```
HashMap<String,String> m;
m = SomeLibrary.foo( );
String s = m.get( "key" );
// can s be null?
```

```
V get( Object key ) {
  if( key == null )
    return getForNullKey( );
  int hash = hash( key.hashCode() );
  for( Entry<K,V> e = table[ indexFor(hash, table.length) ];
                                    e != null; e = e.next ) {
    Object k;
    if( e.hash == hash &&
        ( (k = e.key) == key || key.equals(k) ) )
    return e.value;
  }
  return null;
}
```

Iterate over all entries for this key's hash code

key was not found

Example: Using HashMap (cont'd)

```
HashMap<String,String> m;
m = SomeLibrary.foo( );
if( m.containsKey( "key" ) ) {
  String s = m.get( "key" );
  // can s be null?
  …
}
```

```
V get( Object key ) {
  if( key == null )
    return getForNullKey( );
  int hash = hash( key.hashCode() );
  for( Entry<K,V> e = table[ indexFor(hash, table.length) ];
                                   e != null; e = e.next ) {
    Object k;
    if( e.hash == hash &&
        ( (k = e.key) == key || key.equals(k) ) )
    return e.value;
  }
  return null;
}
```

Is [hash, **null**] a valid entry?
Need to find and check all ways of entering information into table

Example: Maintaining ImageFile

```
class ImageFile {
  String file;
  Image image;

  …

  int hashcode( ) {
    if( image == null )
      return file.hashcode( );
    else
      return image.hashcode( ) + file.hashcode( );
  }
}
```

Is this a suitable implementation of hashcode?

Example: Maintaining ImageFile (cont'd)

```
class ImageFile {
  String file;
  Image image;

  …

  int hashcode( ) {
    if( image == null )
      return file.hashcode( );
    else
      return image.hashcode( ) + file.hashcode( );
  }
}
```

Need to determine whether file may be null

Need to determine whether image may be modified

```
void demo(
    HashMap<ImageFile,String> m,
    ImageFile f ) {
  m.put( f, "Hello" );
  Image i = f.getImage( );
  int l = m.get( f ).length( );
  …
}
```

With lazy initialization, getter may change hash code

Example: Maintaining ImageFile (cont'd)

```
void demo(
    HashMap<ImageFile,String> m,
    ImageFile f ) {
  m.put( f, “Hello” );
  Image i = f.getImage( );
  int l = m.get( f ).length( );
  …
}
```

Hash code is not affected by lazy initialization

```
class ImageFile {
  String file;
  Image image;

  …

  int hashcode( ) {
    return getImage( ).hashcode( ) +
              file.hashcode( );
  }
}
```

Need to determine whether file may be null

Need to determine whether the result of getImage may be modified

ETH zürich

Example: Extending List

```
class SmallList extends List {
  void shrink( ) {
    // reduce array size if the array
    // is not fully used
  }
}
```

Is this an optimization or does it change the behavior?

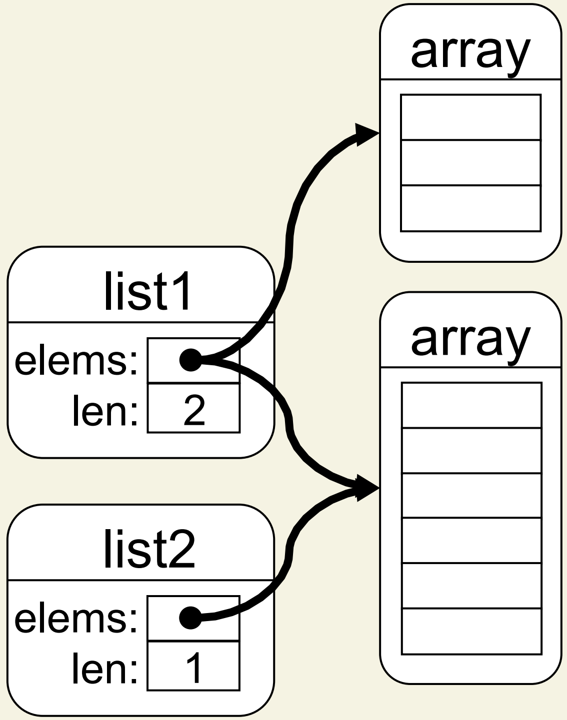
Extending List: Destructive Updates

```
class SmallList extends List {
  void shrink( ) {
    int l = elems.length / 2;
    if( len <= l ) {
      E[ ] tmp = new E[ l ];
      System.arraycopy( elems, 0, tmp, 0, len );
      elems = tmp;
    }
  }
}
```

Is this an optimization or does it change the behavior?

Need to determine whether the elems array may be shared and modified

```
List list2 = list1.take( );
list1.shrink( );
list1.set( 0, "Demo" );
list2.get( 0 );
```



Source Code is Insufficient

- Developers require information that is difficult to extract from source code
  - Possible result values of a method, and when they occur
  - Possible side effects of methods
  - Consistency conditions of data structures
  - How data structures evolve over time
  - Whether objects are shared among data structures
- Details in the source code may be overwhelming

Source Code is Insufficient (cont'd)

- Source code does not express which properties are stable during software evolution
  - Which details are essential and which are incidental?

```
int find( int[ ] array, int v ) {
  for( int i = 0; i < array.length; i++ )
    if( array[ i ] == v )  return i;
  return -1;
}
```

Can we rely on the result r being the smallest index such that array[r] == v?

```
int find( int[ ] array, int v ) {
  if( 256 <= array.length ) {
    // perform parallel search and
    // return first hit
  } else {
    // sequential search like before
  }
}
```

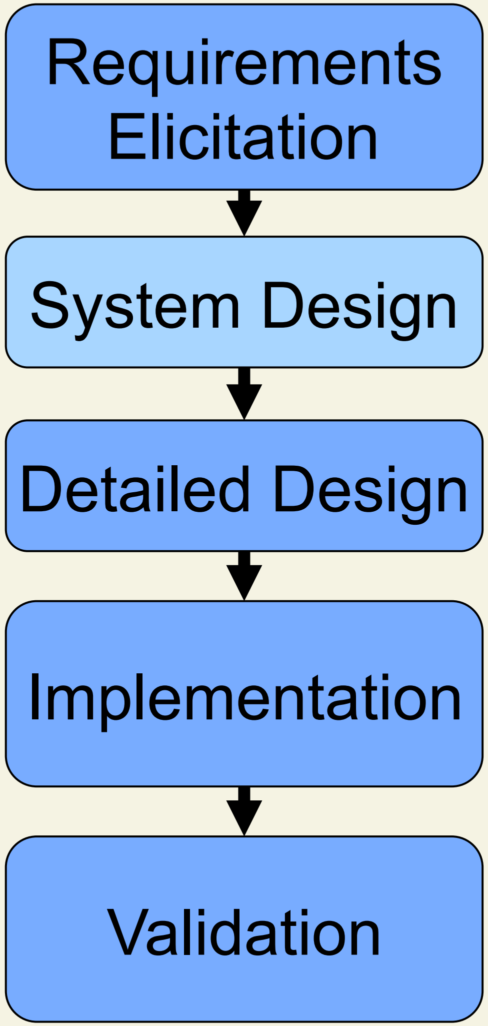
Logistics

- Exercise room issue resolved
  - Check Piazza for exercise & project group assignments

- Strongly encourage everyone to
  - Work on the exercise assignments
  - Attend & actively participate in exercise sessions

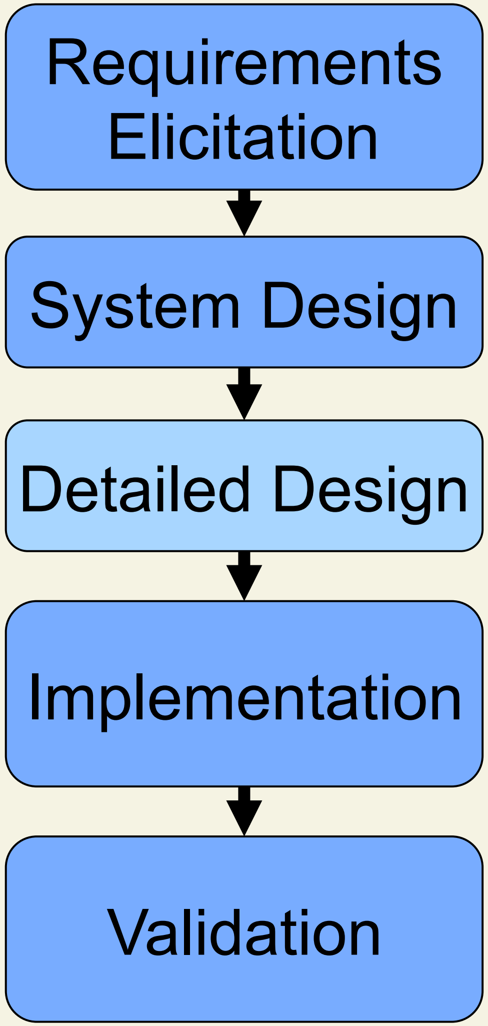
- Feedback always welcomed
  - The class overall
  - Lectures
  - Exercise sessions

System Design



- System design determines the software architecture as a composition of sub-systems
- Components: Computational units with specified interface
  - Filters, databases, layers
- Connectors: Interactions between components
  - Method calls, pipes, events

Detailed Design



- Detailed design
  - Chooses among different ways to implement the system design
  - Provides the basis for the implementation

- Data structures
- Algorithms
- Subclass hierarchies

3. Modeling and Specification

3.1 Code Documentation

3.1.1 What to Document

3.1.2 How to Document

3.2 Informal Models

3.3 Formal Models

Documentation

- Essential properties must be documented explicitly

For clients: How to use the code? Document the interface	For implementers: How does the code work? Document the implementation

- Documentation should focus on what the essential properties are, not how they are achieved
  - "Whenever a List object's shared-field is false, its array is used as representation of at most one List object"

  Rather than

  - "When creating a new List object with an existing array, the shared-field is set to true"

Interface Documentation

For clients:
How to use the code?
Document the interface

- The client interface of a class consists of
  - Constructors
  - Methods
  - Public fields
  - Supertypes
- We focus on methods here
  - Constructors are analogous
  - Fields can be viewed as getter and setter methods

Method Documentation: Call

- Clients need to know how to call a method correctly

```
class InputStreamReader {
  int read( char cbuf[ ], int offset, int len ) throws IOException
  …
}
```

- Parameter values
  - cbuf is non-null
  - offset is non-negative
  - len is non-negative
  - offset + len is at most cbuf.length
- Input state
  - The receiver is open

Method Documentation: Results

- Clients need to know what a method returns

```
class InputStreamReader {
  int read( char cbuf[ ], int offset, int len ) throws IOException
  …
}
```

- Result values
  - The method returns -1 if the end of the stream has been reached before any characters are read
  - Otherwise, the result is between 0 and len, and indicates how many characters have been read from the stream

(Insufficient) Java API Documentation

read

```
public int read(char[] cbuf,
                int offset,
                int length)
         throws IOException
```

Reads characters into a portion of an array.

Specified by:

`read in class Reader`

Parameters:

`cbuf - Destination buffer`

`offset - Offset at which to start storing characters`

`length - Maximum number of characters to read`

Returns:

`The number of characters read, or -1 if the end of the stream has been reached`

Throws:

`IOException - If an I/O error occurs`

Method Documentation: Effects

- Clients need to know how a method affects the state

- Heap effects
  - "result" characters have been consumed from the stream and stored in cbuf, from offset onward
  - If the result is -1, no character is consumed, and cbuf is unchanged

- Other effects
  - The method throws an IOException if the stream is closed or an I/O error occurs
  - It does not block

Method Documentation: Another Example

```
class List<E> {
  …
  List<E> clone( ) {
    return new List<E>( elems.clone( ), len );
  }
}
```

- The method returns a shallow copy of its receiver
  - The list is copied, but not its contents
- The result is a fresh object
- The method requires constant time and space

Interface Documentation: Global Properties

- Some implementations have properties that affect all methods
  - Properties of the data structure, that is, guarantees that are maintained by all methods together
  - Requirements made by all methods

- Consistency: properties of states
  - Example: a list is sorted
  - Gives guarantees for various methods
  - Client-visible invariants

```
int a = list.first( );
int b = list.get( 1 );
int c = list.last( );
// a <= b <= c
```

Interface Document.: Global Properties (cont'd)

- Evolution: properties of sequences of states
  - Example: a list is immutable
  - Gives guarantees for various methods
  - Invariants on sequences of states

```
int a = list.first( );
// arbitrary operations
int b = list.first( );
// a == b
```

- Abbreviations: requirements or guarantees for all methods
  - Example: a list is not thread-safe
    Clients must ensure they have exclusive access to the list, for instance, because the execution is sequential, the list is thread-local, or they have acquired a lock

Implementation Documentation

For implementers:
How does the code work?
Document the implementation

- Method documentation is similar to interfaces
  - Often more details, for instance, effects on fields
  - Includes hidden methods
- Data structure documentation is more prominent
  - Properties of fields, internal sharing, etc.
  - Implementation invariants
- Documentation of the algorithms inside the code
  - For instance, justification of assumptions

Implementation Documentation: Example

```
class List<E> {
  E[ ] elems;
  int len;
  boolean shared;

  …
}
```

1. elems is non-null
2. When the shared-field is true then the elems-array is immutable
3. When the shared-field is false, the elems-array is used as representation of at most one List object
4. elems is pointed to only by List objects
5. 0 <= len <= elems.length

Impl. Documentation: Example (cont'd)

```
/* This method reduces the memory footprint of the list if it uses at most
 * 50% of its capacity, and does nothing otherwise. It optimizes the
 * memory consumption if the underlying array is not shared or if it is
 * shared but will be copied several times after shrinking. The list content
 * remains unchanged. */
void shrink( ) {
  // perform array copy only if array size can be reduced by 50%
  int l = elems.length / 2;
  if( len <= l ) {
    E[ ] tmp = new E[ l ];
    System.arraycopy( elems, 0, tmp, 0, len );
    elems = tmp;
    shared = false;
  }
}
```

Impl. Documentation: Example (cont'd)

```
void shrink( ) {
  int l = elems.length / 2;
  if( len <= l ) {
    E[ ] tmp = new E[ l ];
    System.arraycopy( … );
    elems = tmp;
    shared = false;
  }
}
```

1. elems is non-null
2. When the shared-field is true then the elems-array is immutable
3. When the shared-field is false, the elems-array is used as representation of at most one List object
4. elems is pointed to only by List objects
5. 0 <= len <= elems.length

Documentation: Key Properties

- Methods and constructors
  - Arguments and input state
  - Results and output state
  - Effects
- Data structures
  - Value and structural invariants
  - One-state and temporal invariants
- Algorithms
  - Behavior of code snippets (analogous to methods)
  - Explanation of control flow
  - Justification of assumptions

For clients:
How to use the code?
Document the interface

For implementors:
How does the code work?
Document the implementation

3. Modeling and Specification

3.1 Code Documentation

3.1.1 What to Document

3.1.2 How to Document

3.2 Informal Models

3.3 Formal Models

Comments

- Simple, flexible way of documenting interfaces and implementations
- Tool support is limited
  - HTML generation
  - Not present in executable code
  - Relies on conventions
- Javadoc
  - Textual descriptions
  - Tags

```
/**
 * Returns the value to which the
 * specified key is mapped, or
 * {@code null} if this map contains no
 * mapping for the key.
 *
 * @param key the key whose associated
 *          value is to be returned
 * @return the value to which the
 *          specified key is mapped, or
 *          {@code null} if this map contains
 *          no mapping for the key
 * @throws NullPointerException if the
 *          specified key is null and this map
 *          does not permit null keys
 */
V get( Object key );
```

Types and Modifiers

- Types document typically syntactic aspects of inputs, results, and invariants

```
HashMap<String,String> m;
m = SomeLibrary.foo( );
String s = m.get( “key” );
```

```
from SomeLibrary import foo
m = foo( )
s = m[ 'key' ]
```

Python

- Modifiers can express some specific semantic properties
- Tool support
  - Static checking
  - Run-time checking
  - Auto-completion

```
class HashMap<K,V> … {
  final float loadFactor;
  …
}
```

Effect Systems

- Effect systems are extensions of type systems that describe computational effects
  - Read and write effects
  - Allocation and de-allocation
  - Locking
  - Exceptions

```
class InputStreamReader {
  int read( ) throws IOException
  …
}
```

```
try {
  int i = isr.read( );
} catch( IOException e ) {
  …
}
```

- Tool support
  - Static checking
- Trade-off between overhead and benefit

Metadata

- Annotations allow one to attach additional syntactic and semantic information to declarations
- Tool support
  - Type checking of annotations
  - Static processing through compiler plug-ins
  - Dynamic processing

```
@interface NonNull{ }
```

```
@NonNull Image getImage( ) {
  if( image == null ) {
    // load the image
  }
  return image;
}
```

```
@interface UnderConstruction {
  String owner( );
}
```

```
@UnderConstruction(
            owner = "Busy Guy" )
class ResourceManager { … }
```

Assertions

- Assertions specify semantic properties of implementations
  - Boolean conditions that need to hold

- Tool support
  - Run-time checking
  - Static checking
  - Test case generation

```
void set( int index, E e ) {
  if( shared ) {
    elems = elems.clone( );
    shared = false;
  }
  assert !shared;
  elems[ index ] = e;
}
```

Contracts

- Contracts are stylized assertions for the documentation of interfaces and implementations
  - Method pre and postconditions
  - Invariants
- Tool support
  - Run-time checking
  - Static checking
  - Test case generation

```
class ImageFile {
  String file;
  invariant file != null;

  Image image;
  invariant old( image ) != null  ==>
                old( image ) == image;

  ImageFile( String f )
    requires f != null;
  { file = f; }

  Image getImage( )
    ensures result != null;
  {
    if( image == null ) { // load the image }
    return image;
  }
}
```

Documentation: Techniques

- Trade-off between overhead, expressiveness, precision, and benefit
  - Formal techniques require more overhead, but enable better tool support
  - In practice, a mix of the different techniques is useful

- It is better to simplify than to describe complexity!
  - *If you have a procedure with ten parameters, you probably missed some.* [Alan J. Perlis]

3. Modeling and Specification

3.1 Source Code

3.2 Informal Models

3.3 Formal Models

Underspecification

- Software is typically designed iteratively
- Each iteration adds details and reflects design decisions that have been left open in the previous iteration
  - Choice of data structures
  - Choice of algorithms
  - Details of control and data flow

```
class University {
  Set<Student> students;
  …
}
```

```
class Student {
  Program major;
  …
}
```

```
class University {
  Map<Student, Program> enrollment;
  …
}
```

```
class Student {
  …
}
```

Underspecification (cont'd)

- Dispatch an event to all observers

```
class Subject {
  Set<Observer> observers;

  /* This method calls update
   * on each registered observer
   * in an unspecified order.
   */
  void notify( ) {
    for( Observer o : observers )
      o.update( );
  }
}
```

- Open bank account if all conditions are met

```
abstract class Account {
  boolean open;
  abstract boolean
                 allConditions( … );
  void open( … ) {
    if( allConditions( … ) )
      open = true;
    else
      throw …;
  }
}
```

Views

- Many software engineering tasks require specific views on the design

- Examples
  - Software architecture: Is it possible for an app to be terminated without prior notification?
  - Test data generation: What are all the possible object configurations for a data structure?
  - Security review: What is the communication protocol between a client and the server?
  - Deployment: Which software component runs on which hardware?

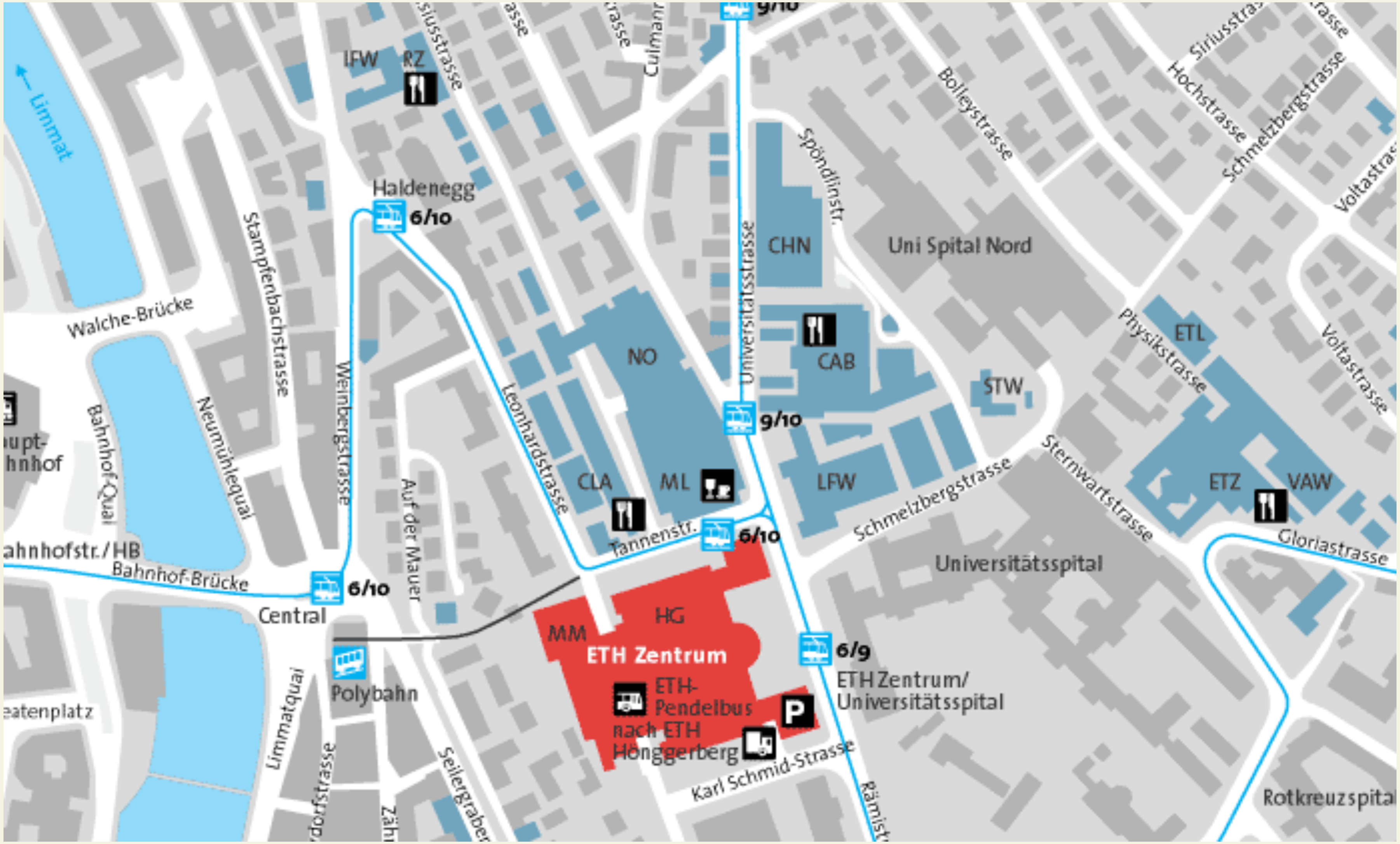
Design Specifications

- Source code provides very limited support for leaving design choices unspecified
  - Often because code is executable
  - In some cases, subclassing can be used
- Some relevant design information is not represented in the program or difficult to extract
  - Source code and documentation are too verbose
  - Tools can extract some information like control or data flow graphs
- Design specifications are models of the software system that provide suitable abstractions

What is Modeling?

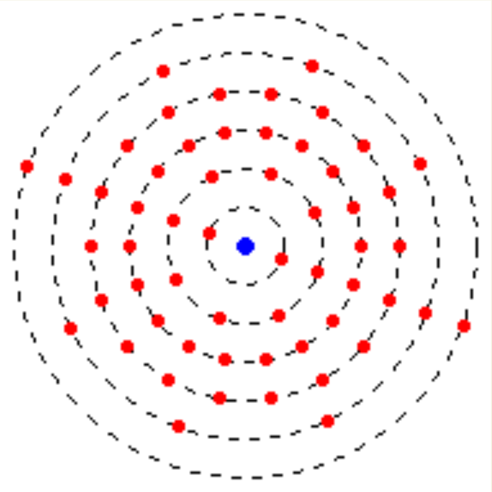
- Building an abstraction of reality
  - Abstractions from things, people, and processes
  - Relationships between these abstractions
- Abstractions are simplifications
  - They ignore irrelevant details
  - What is relevant or irrelevant depends on the purpose of the model
- Draw complicated conclusions in the reality with simple steps in the model
- Modeling is a means for dealing with complexity

Example 1: Street Map

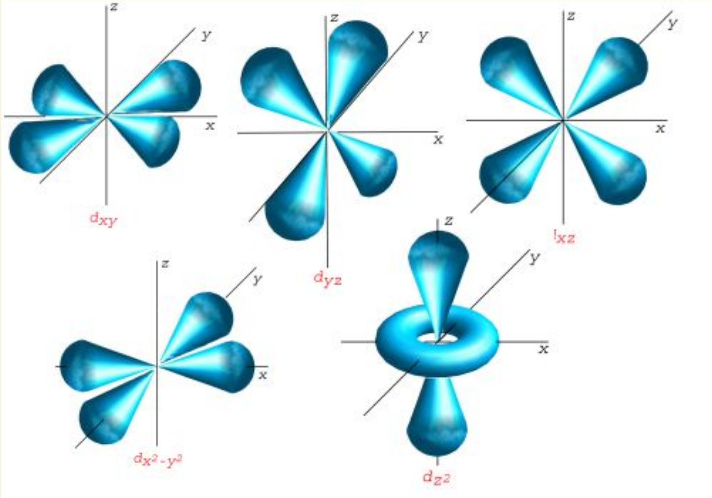


Example 2: Atom Models in Physics

- Bohr model
  - Nucleus surrounded by electrons in orbit
  - Explains, e.g., spectra



- Quantum physics
  - Position of electrons described by probability distribution
  - Takes into account Heisenberg's uncertainty principle



Logistics

- Project 1, part 1 posted (due March 15th)
  - Start early on this with your team members

- Exercise 2 posted
  - Please work on it

- Exercise 1 sample solution posted
  - Attend the exercise sessions for Q/A, etc.

The Unified Modeling Language UML

- UML is a modeling language
  - Using text and graphical notation
  - For documenting specification, analysis, design, and implementation
- Importance
  - Recommended OMG (Object Management Group) standard notation
  - De facto standard in industrial software development

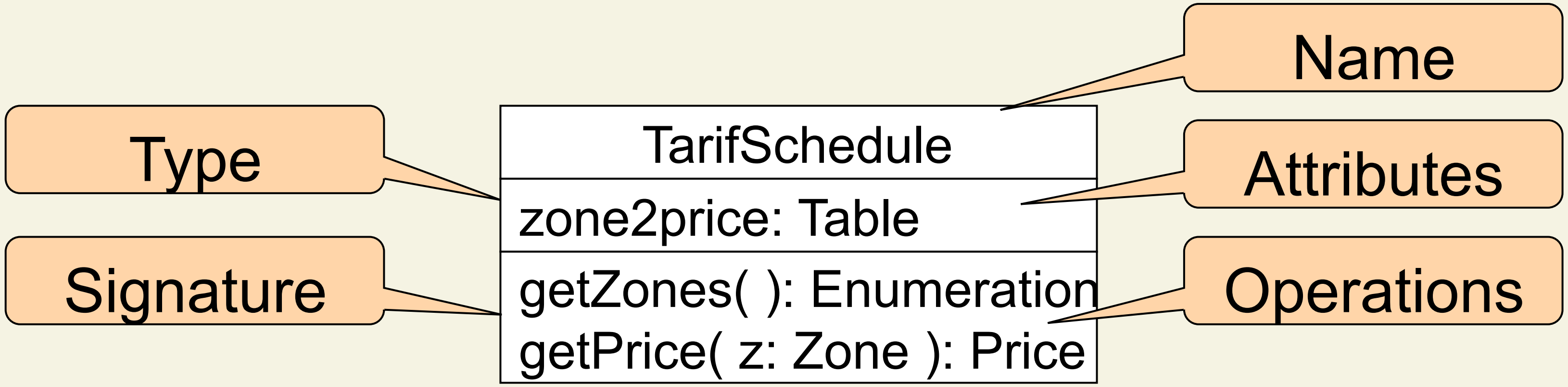
UML Notations

- Use case diagrams – requirements of a system
- Class diagrams – structure of a system
- Interaction diagrams – message passing
  - Sequence diagrams
  - Collaboration diagrams
- State and activity diagrams – actions of an object
- Implementation diagrams
  - Component model – dependencies between code
  - Deployment model – structure of the runtime system
- Object constraint language (OCL)

3. Modeling and Specification

3.1 Code Documentation
3.2 Informal Models
3.2.1 Static Models
3.2.2 Dynamic Models
3.2.3 Contracts
3.2.4 Mapping Models to Code
3.3 Formal Models

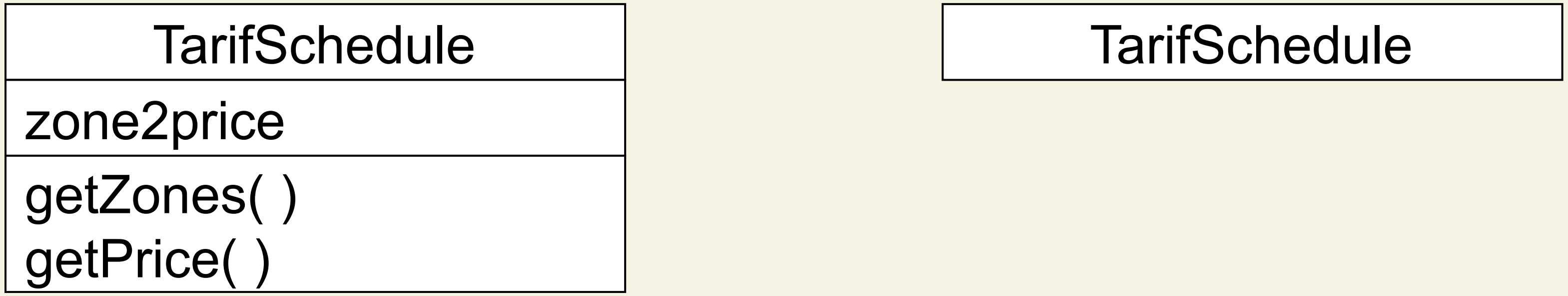
Classes



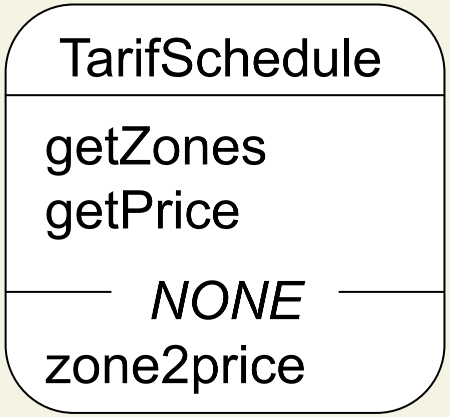
- A class includes state (attributes) and behavior (operations)
  - Each attribute has a type
  - Each operation has a signature
- The class name is the only mandatory information

More on Classes

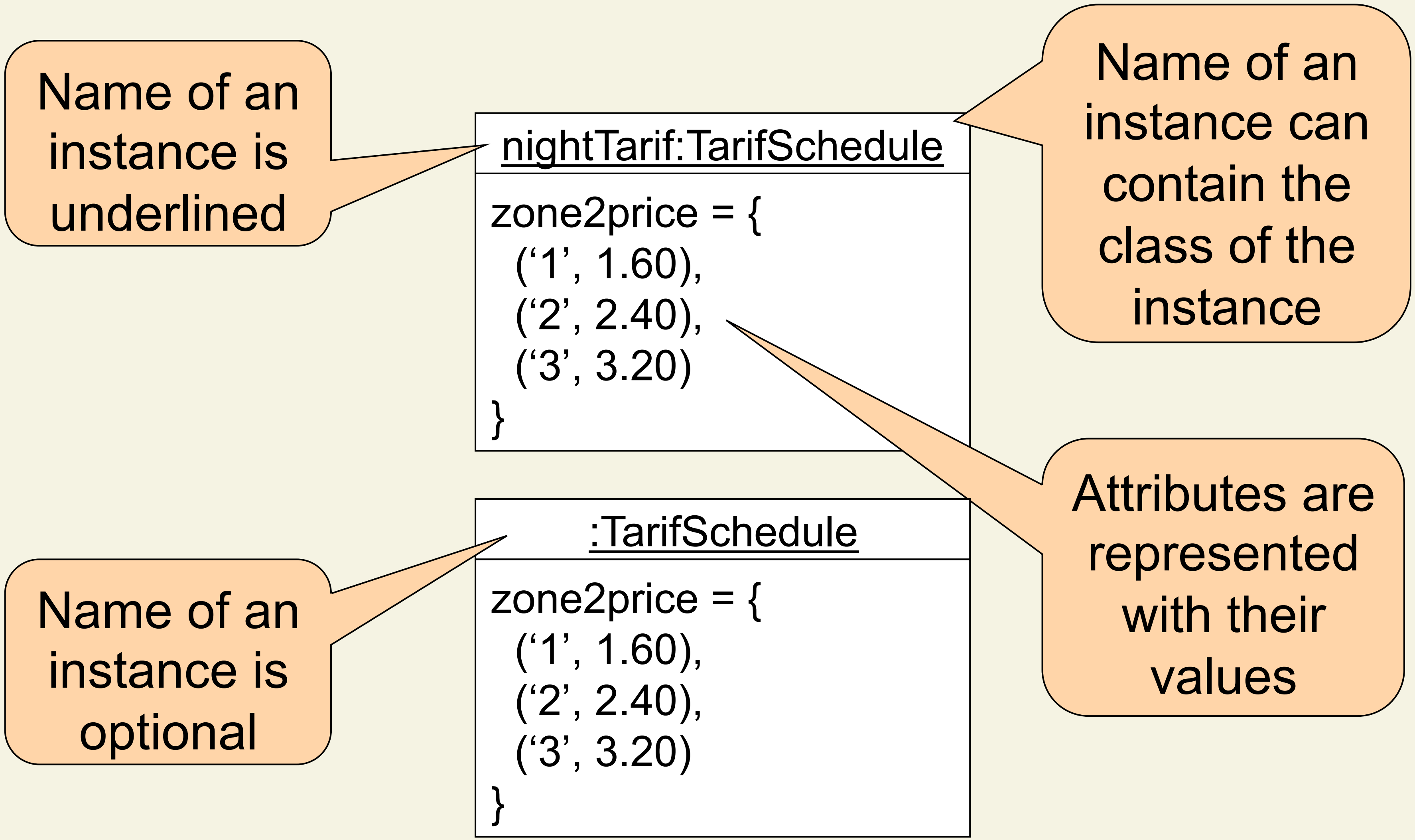
- Valid UML class diagrams



- Corresponding BON diagram
  - No distinction between attributes and operations (uniform access principle)

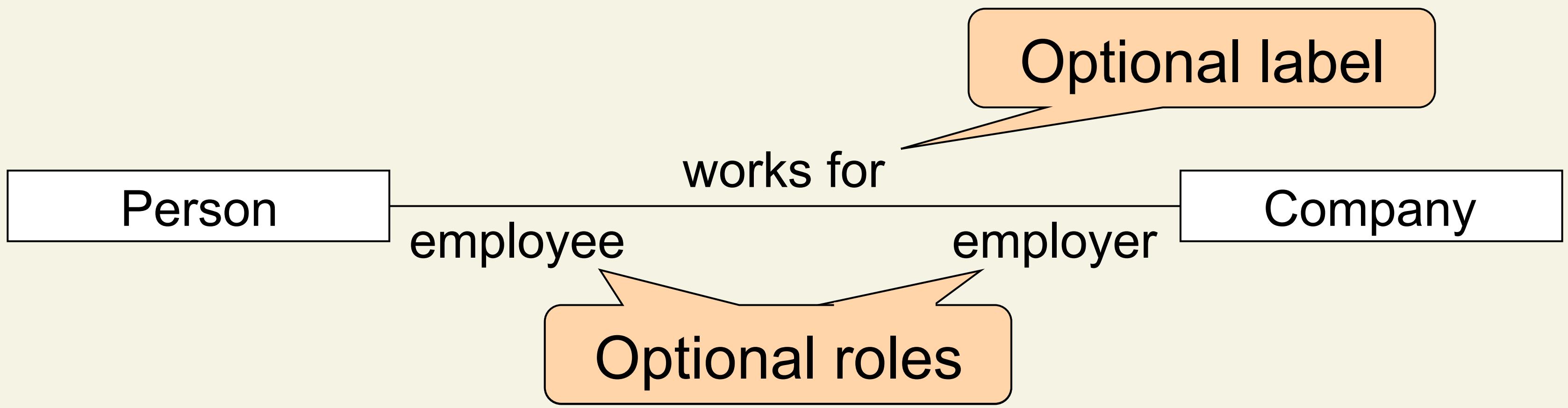


Instances (Objects)



Associations

- A link represents a connection between two objects
  - Ability of an object to send a message to another object
  - Object A has an attribute whose value is B
  - Object A creates object B
  - Object A receives a message with object B as argument
- Associations denote relationships between classes

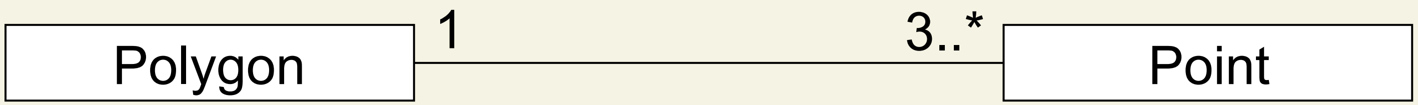


Multiplicity of Associations

- The multiplicity of an association end denotes how many objects the source object can reference
  - Exact number: 1, 2, etc. (1 is the default)
  - Arbitrary number: * (zero or more)
  - Range: 1..3, 1..*

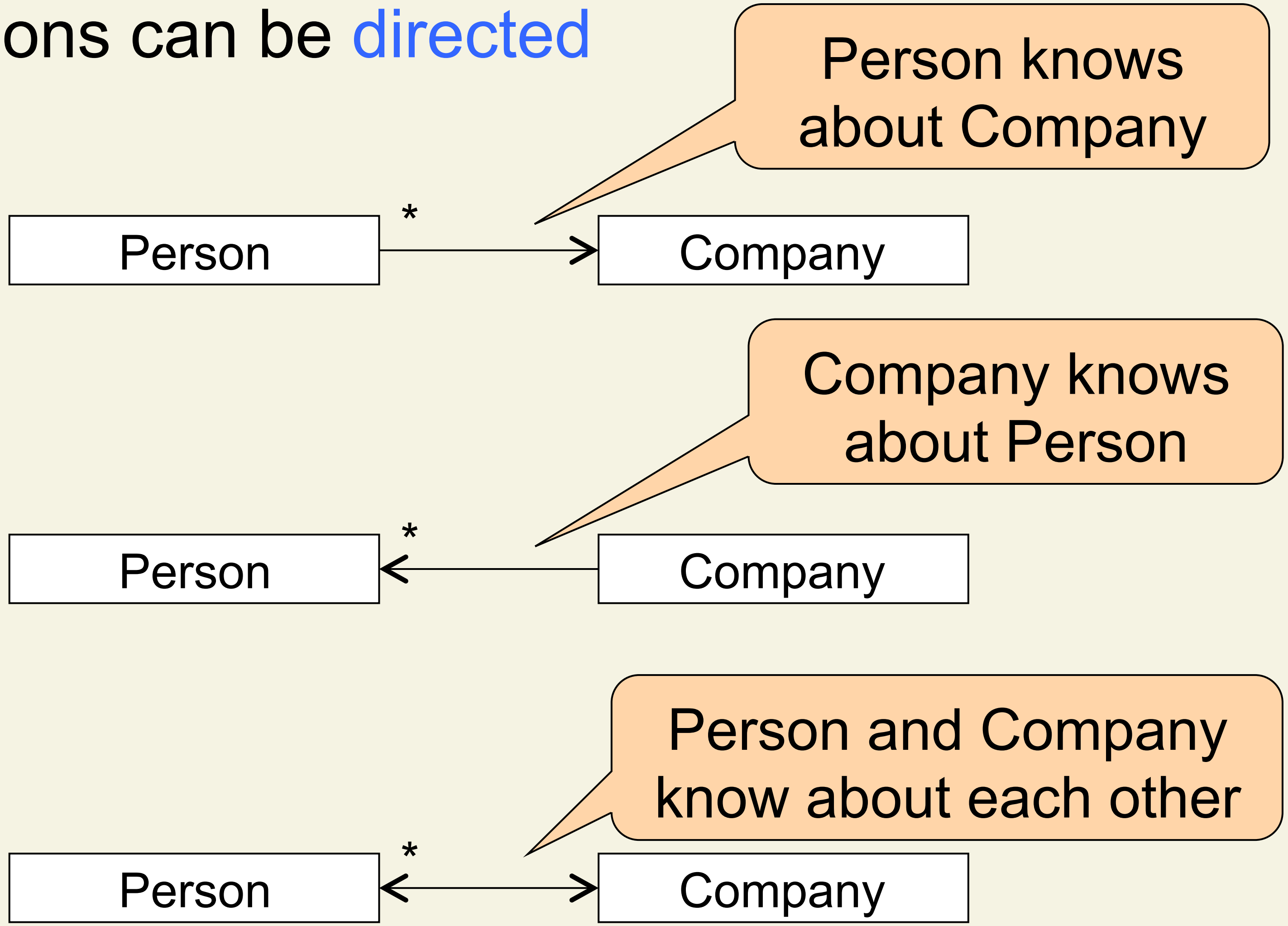
- 1-to-(at most) 1 association

- 1-to-many association



Navigability

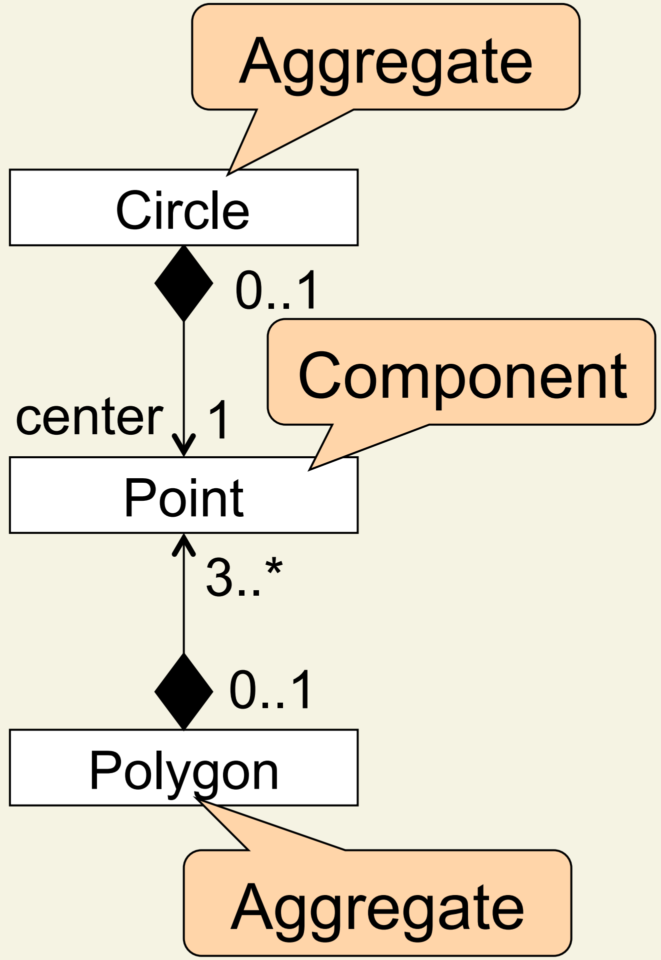
- Associations can be directed



Composition

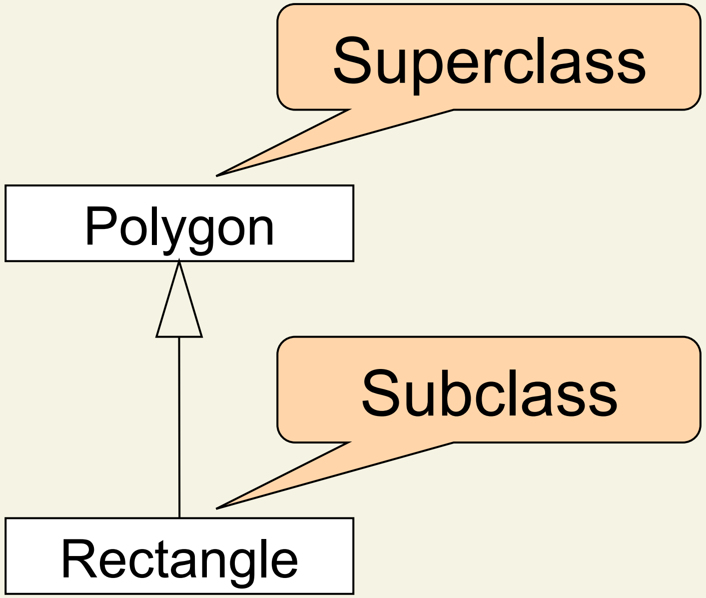
- Composition expresses an exclusive part-of ("has-a") relationship
  - Special form of association
  - No sharing

- Composition can be decorated like other associations
  - Multiplicity, label, roles



Generalization and Specialization

- Generalization expresses a kind-of (“is-a”) relationship
- Generalization is implemented by inheritance
  - The child classes inherit the attributes and operations of the parent class
- Generalization simplifies the model by eliminating redundancy



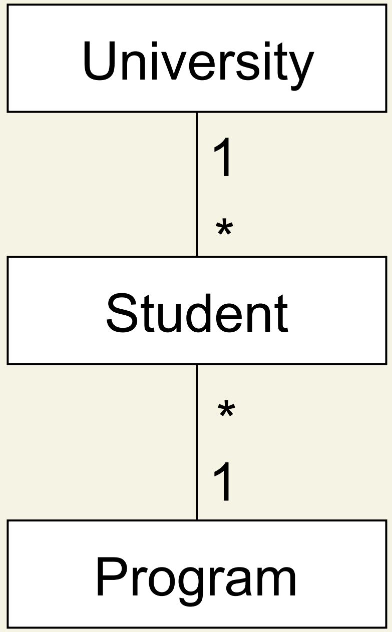
Example: Underspecification

```
class University {
  Set<Student> students;
  …
}
```

```
class Student {
  Program major;
  …
}
```

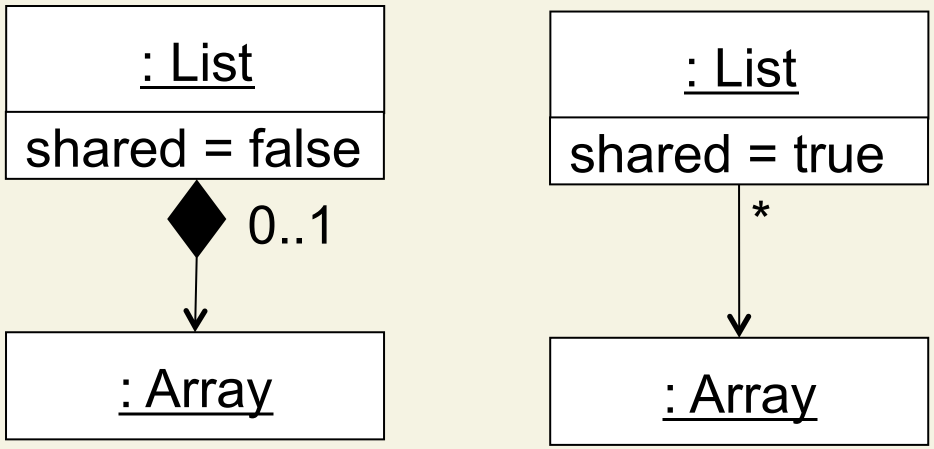
```
class University {
  Map<Student, Program> enrollment;
  …
}
```

```
class Student {
  …
}
```



- The class diagram leaves the choice of data structure unspecified

Example: Views



- The class diagram represents only the structure of the system, not the dynamic behavior
- Some relevant invariants are represented

3. Modeling and Specification

3.1 Code Documentation

3.2 Informal Models

- 3.2.1 Static Models
- 3.2.2 Dynamic Models
- 3.2.3 Contracts
- 3.2.4 Mapping Models to Code

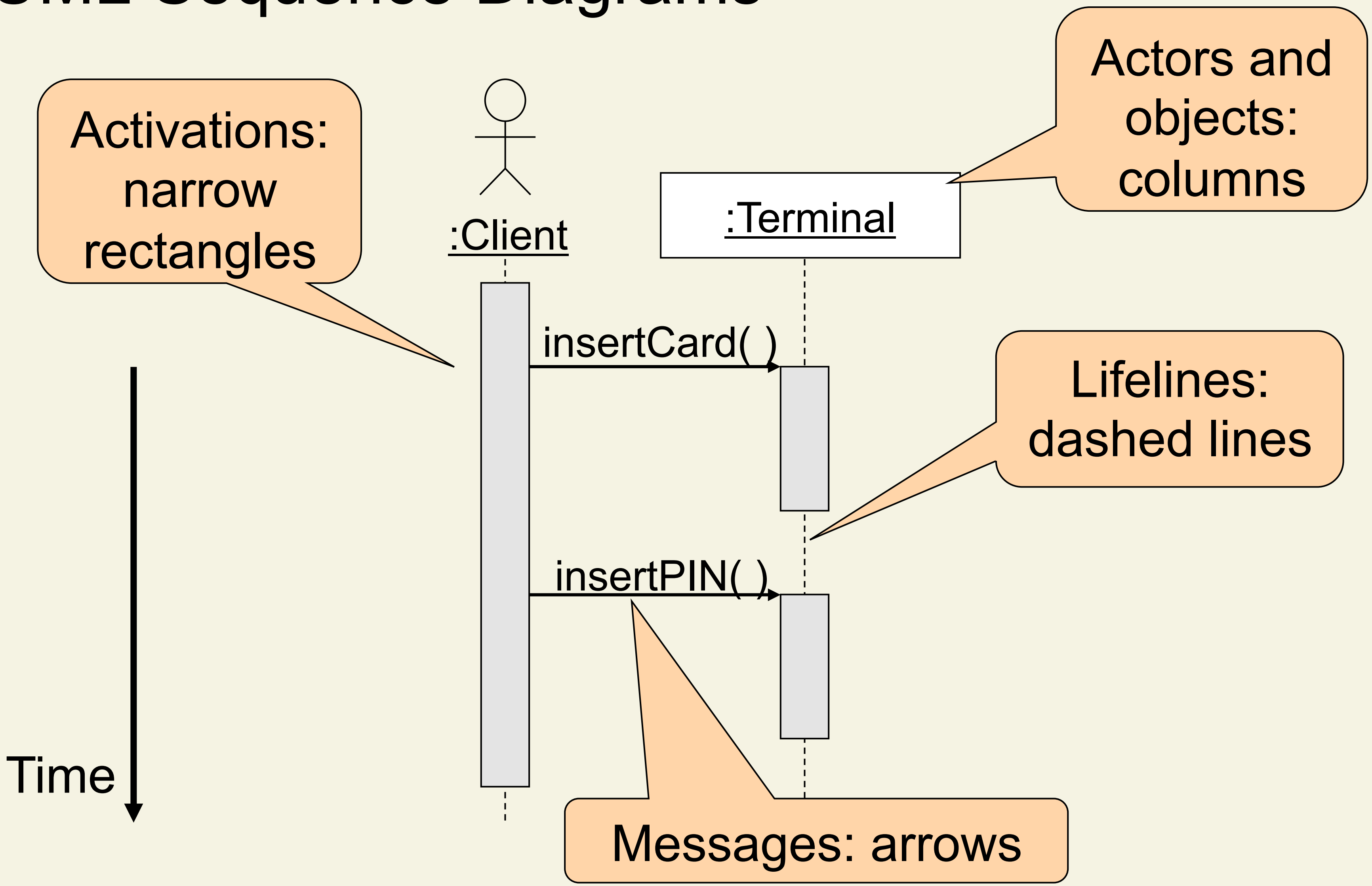
3.3 Formal Models

Dynamic Models

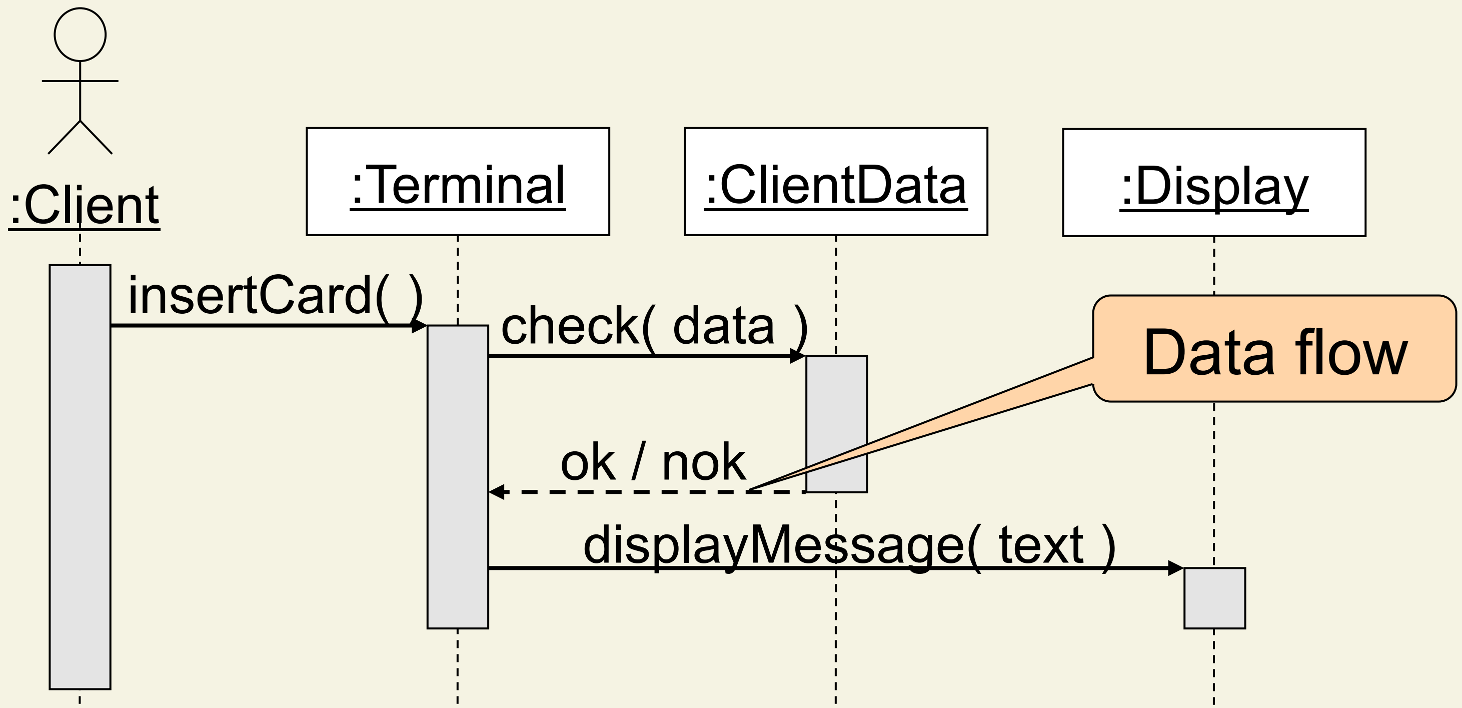
- Static models describe the structure of a system
- Dynamic models describe its behavior

Sequence diagrams describe collaboration between objects	State diagrams describe the lifetime of a single object

UML Sequence Diagrams

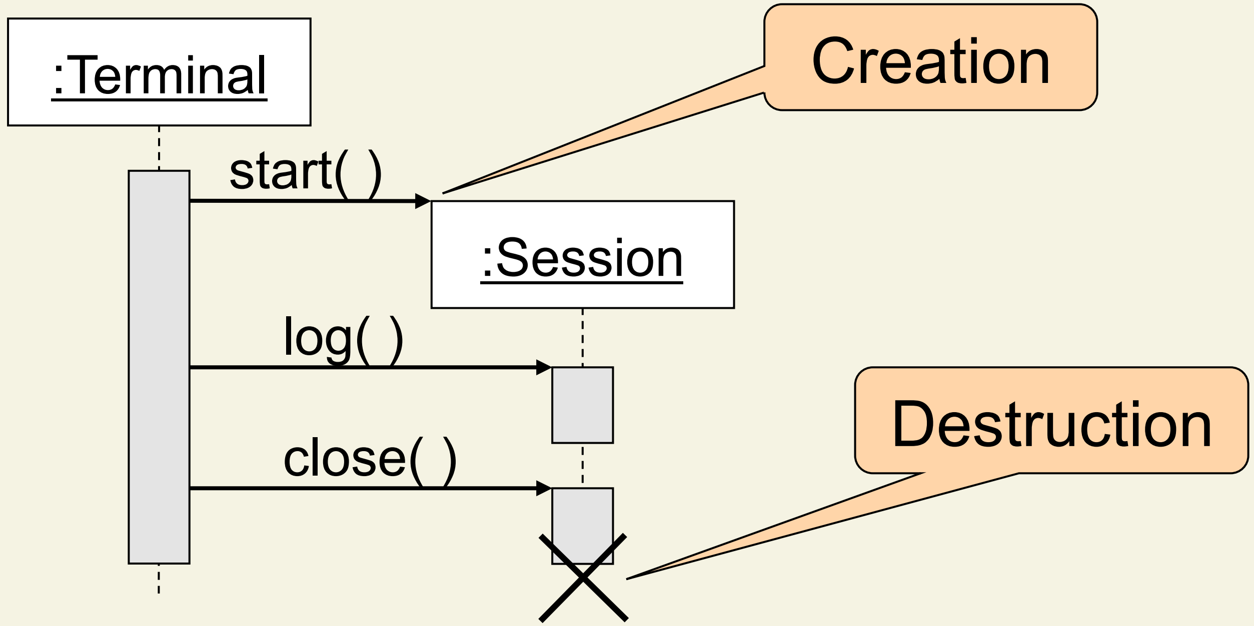


Nested Messages



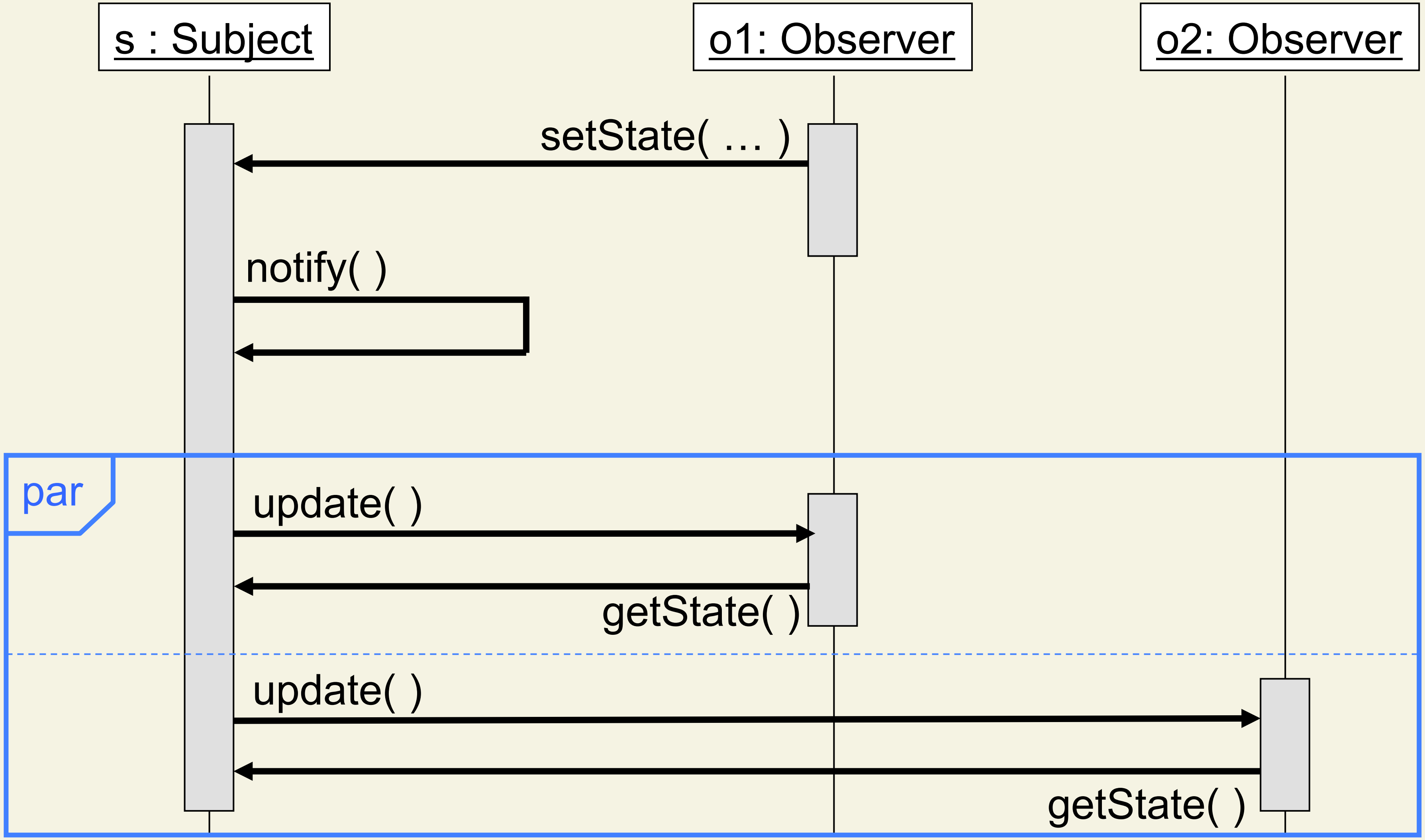
- The source of an arrow indicates the activation which sent the message
- An activation is as long as all nested activations

Creation and Destruction

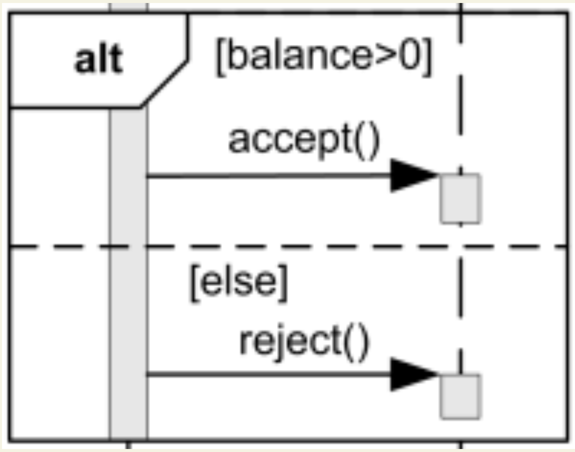


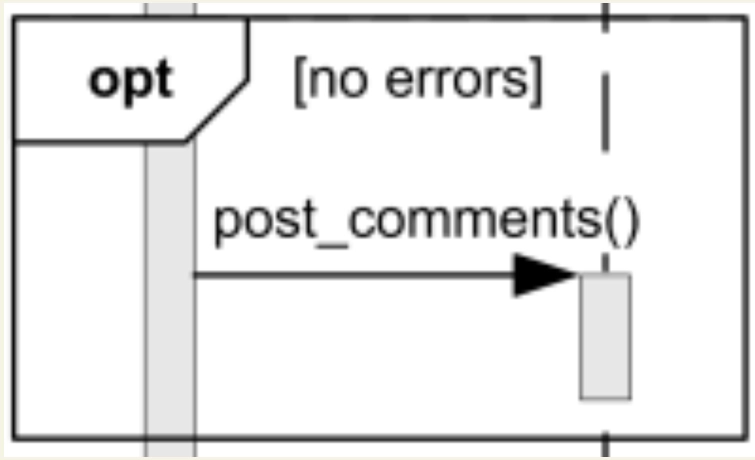
- Creation is denoted by a message arrow pointing to the object
- In garbage collection environments, destruction can be used to denote the end of the useful life of an object

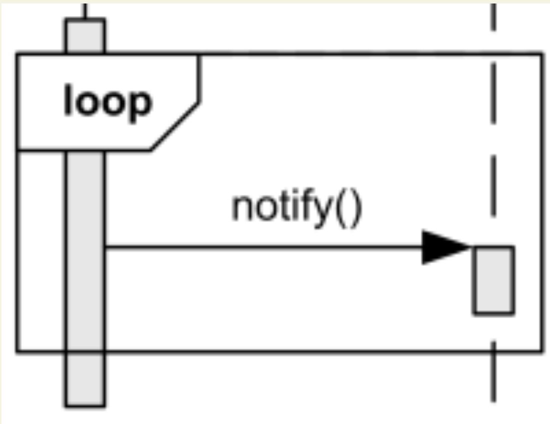
Example: Underspecification and Views

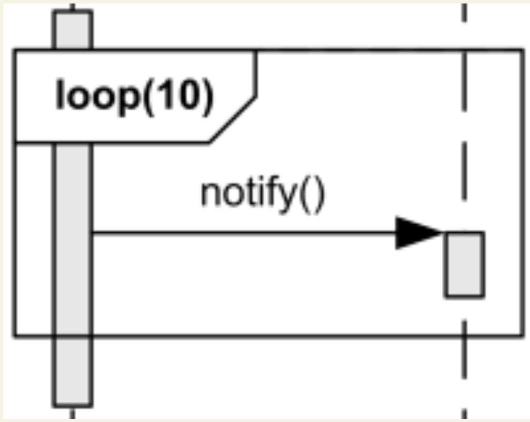


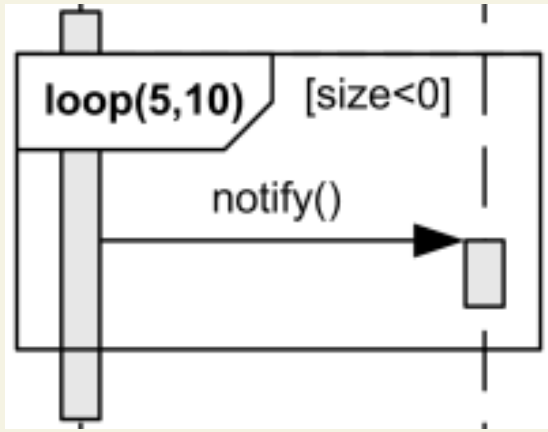
UML Combined Fragments (samples only)









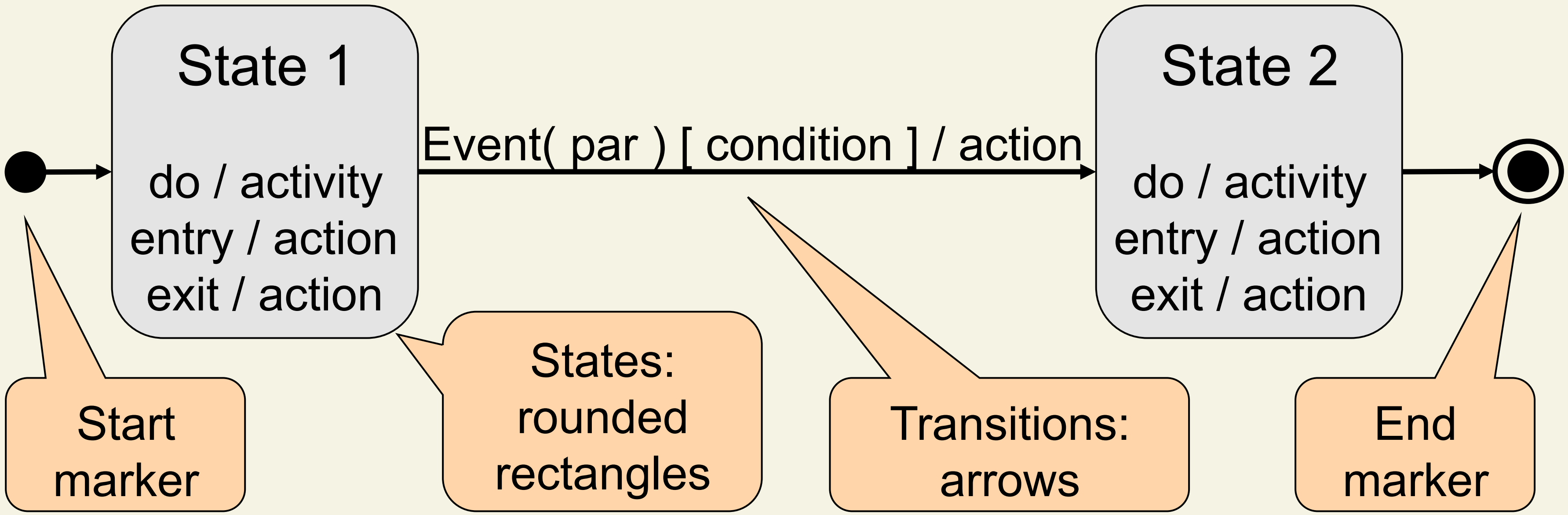


State

- An abstraction of the attribute values of an object
- A state is an equivalence class of all those attribute values and links that do not need to be distinguished for the control structure of the class
- Example: State of an account
  - An account is open, closed, or pending
  - Omissions: account number, owner, etc.
  - All open accounts are in the same equivalence class, independent of their number, owner, etc.

UML State Diagrams

- Objects with extended lifespan often have state-dependent behavior
- Modeled as state diagram (also called state chart)

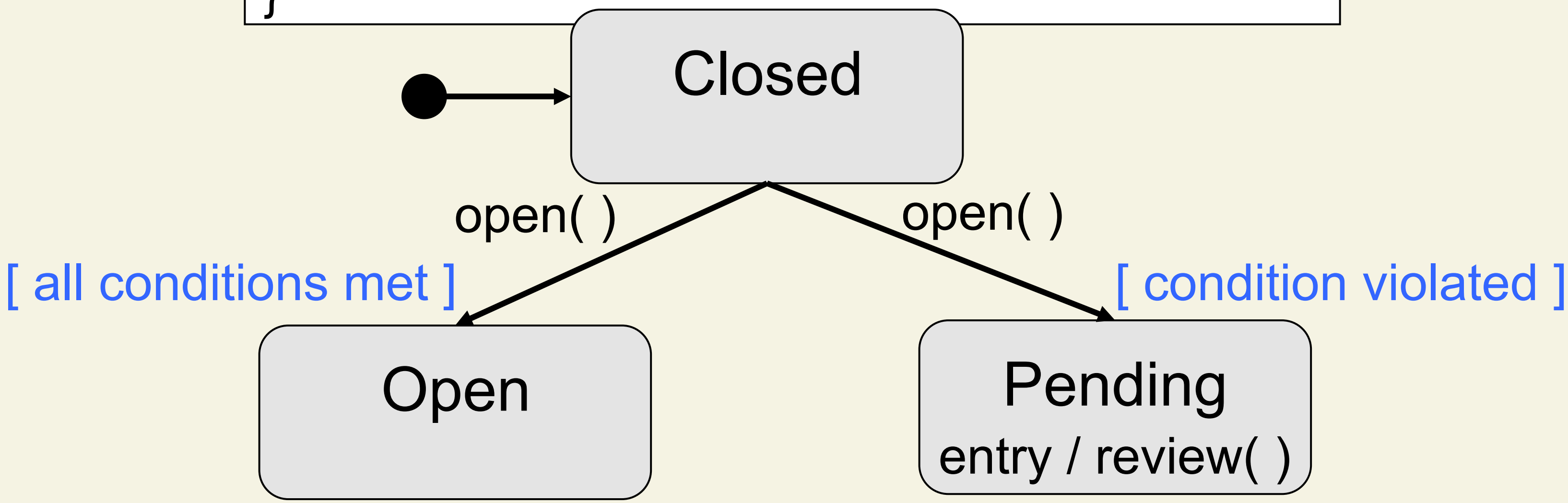


Events, Actions, and Activities

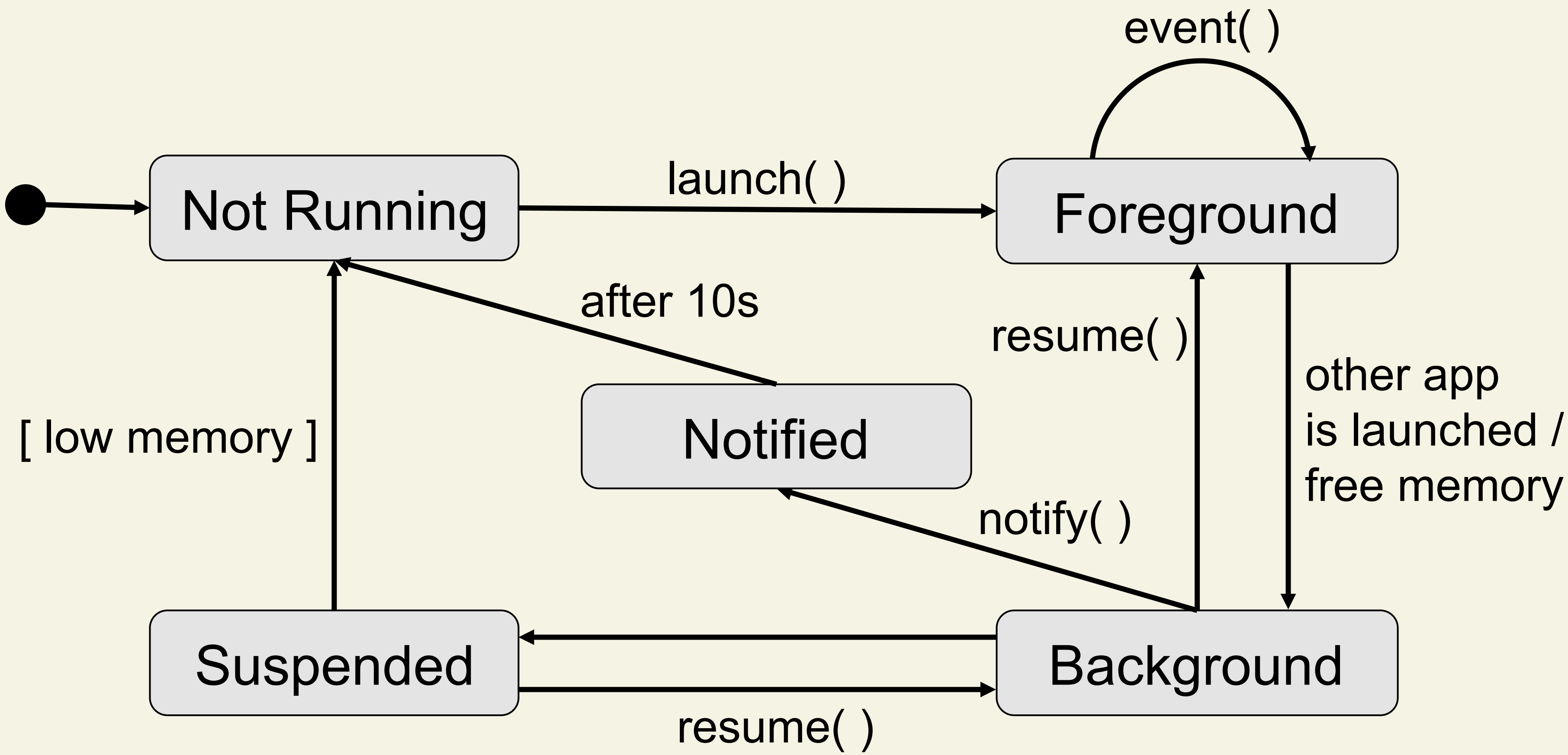
- Event: Something that happens at a point in time
  - Examples: Receipt of a message, change event for a condition, time event
- Action: Operation in response to an event
  - Example: Object performs a computation upon receipt of a message
- Activity: Operation performed as long as object is in some state
  - Example: Object performs a computation without external trigger

Example: Underspecification

```
abstract class Account {
  boolean open;

  abstract boolean allConditions( … );

  void open( … ) {
    if( allConditions( … ) )     open = true;
    else                         throw …;
  }
}
```



Example: Views



Practical Tips for Dynamic Modeling

- Construct dynamic models only for classes with significant dynamic behavior
- Consider only relevant attributes
  - Use abstraction
- Look at the granularity of the application when deciding on actions and activities

3. Modeling and Specification

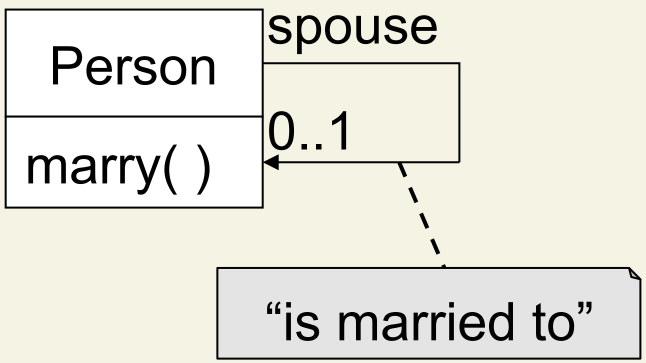
3.1 Code Documentation

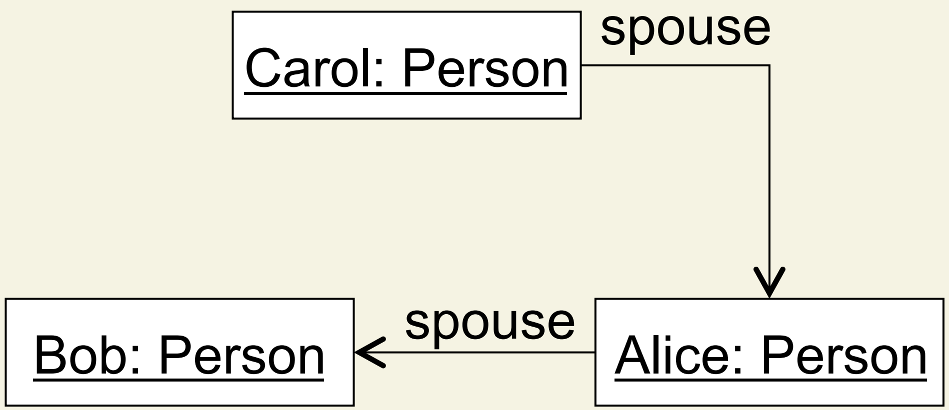
3.2 Informal Models

- 3.2.1 Static Models
- 3.2.2 Dynamic Models
- 3.2.3 Contracts
- 3.2.4 Mapping Models to Code

3.3 Formal Models

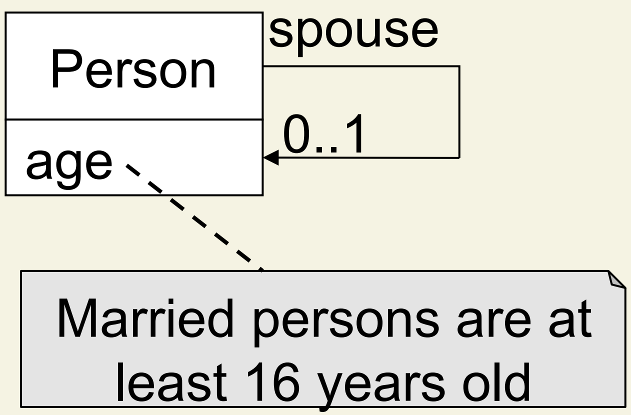
Diagrams are not Enough

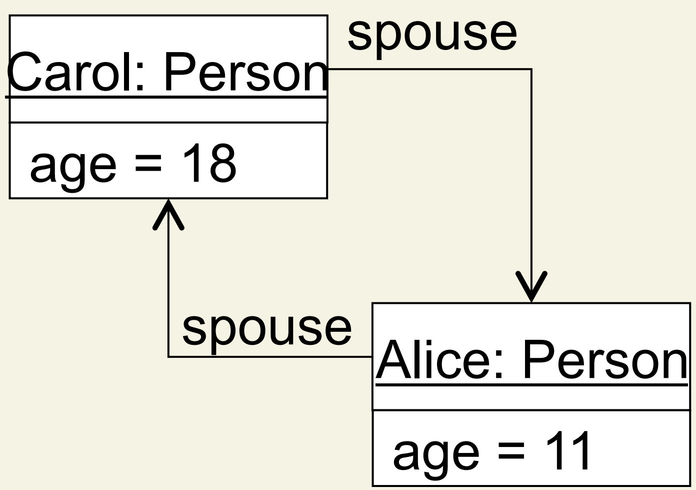




- Carol is married to Alice, Alice is married to Bob, and Bob is not married at all
- A valid instantiation of the class diagram!
- Associations describe relations between classes

Diagrams are not Enough (cont'd)





- Carol is married to Alice, who is only eleven
- A valid instantiation of the class diagram!
- Class diagrams do not restrict values of attributes

Object Constraint Language – OCL

- The contract language for UML
- Used to specify
  - Invariants of objects
  - Pre- and postconditions of operations
  - Conditions (for instance, in state diagrams)
- Special support for
  - Navigation through UML class diagram
  - Associations with multiplicities

Form of OCL Invariants

- Constraints can mention
  - **self**: the contextual instance
  - Attributes and role names
  - Side-effect free methods (stereotype <<query>>)
  - Logical connectives
  - Operations on integers, reals, strings, sets, bags, sequences
  - Etc.

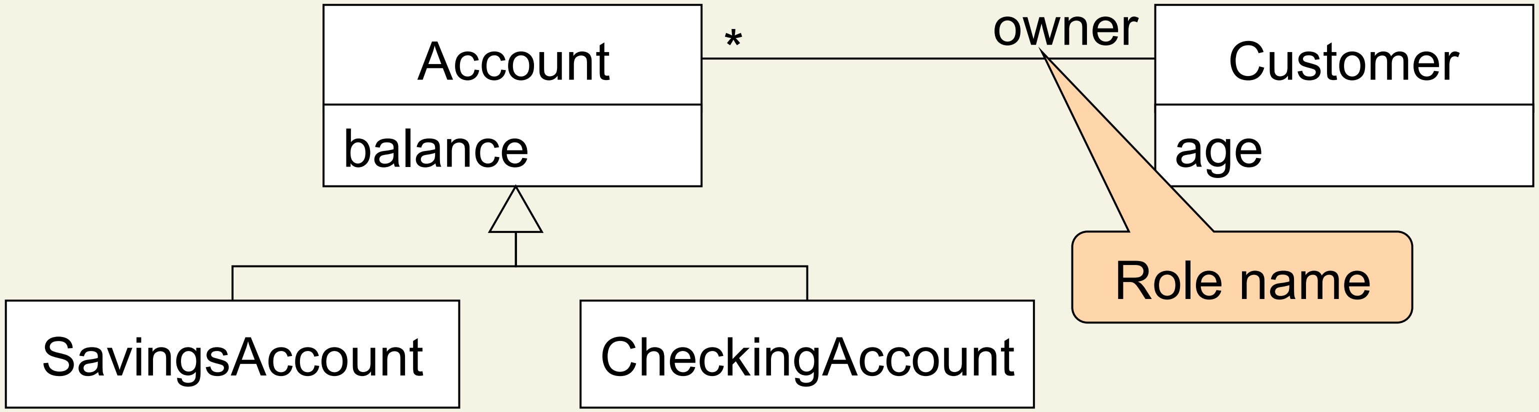
The context is an instance of a class in the UML diagram

Declares an invariant

context Person **inv**:
self.age >= 0

A boolean constraint

OCL Invariants



- A savings account has a non-negative balance

context SavingsAccount **inv**:
self.balance >= 0

- Checking accounts are owned by adults

context CheckingAccount **inv**:
self.owner.age >= 18

OCL Pre- and Postconditions

Context specifies method signature

context Account::Withdraw(a: int)
pre: a >= 0
post: GetBalance( ) = GetBalance@pre( ) - a

Suffix @pre is used to refer to prestate values

3. Modeling and Specification

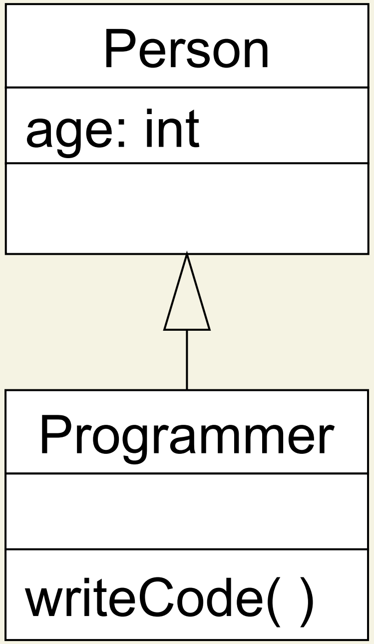
3.1 Code Documentation

3.2 Informal Models

- 3.2.1 Static Models
- 3.2.2 Dynamic Models
- 3.2.3 Contracts
- 3.2.4 Mapping Models to Code

3.3 Formal Models

Implementation of UML Models in Java




```java
class Person {
  private int age;

  public void setAge( int a )
    { age = a; }
  public int getAge( )
    { return age; }
}
```

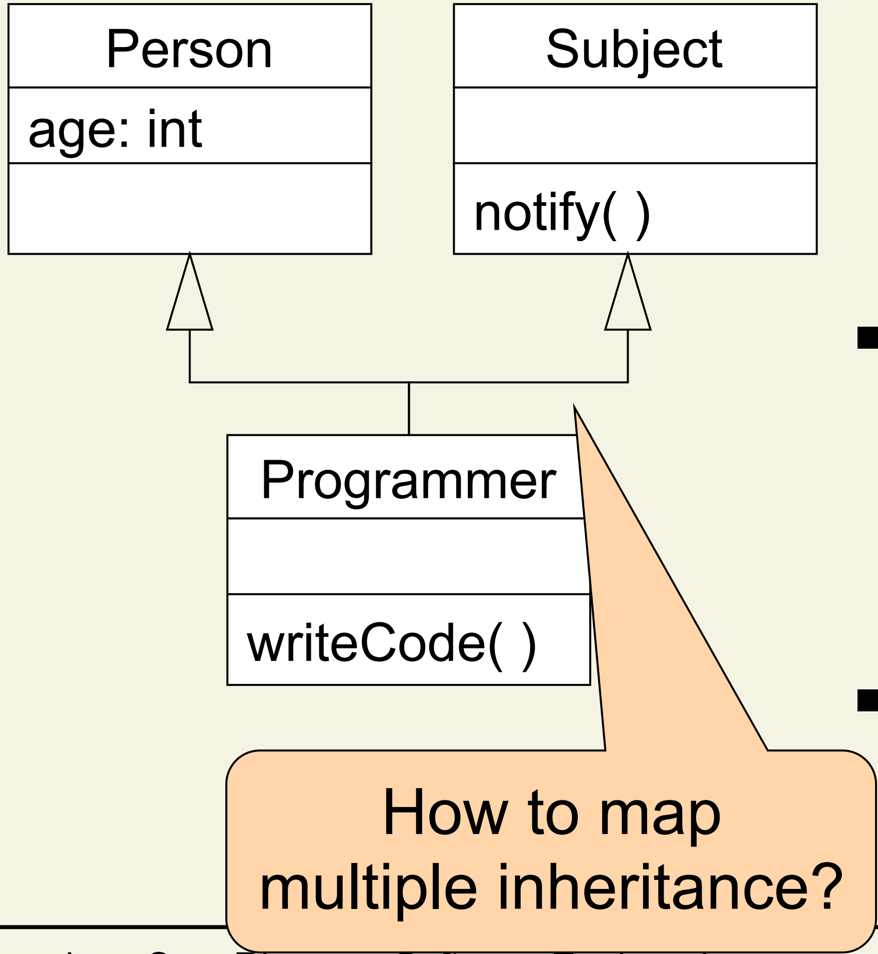
```java
class Programmer extends Person {
  public void writeCode( )
    { … }
}
```

Model-Driven Development: Idea

- Work on the level of design models
- Generate code automatically

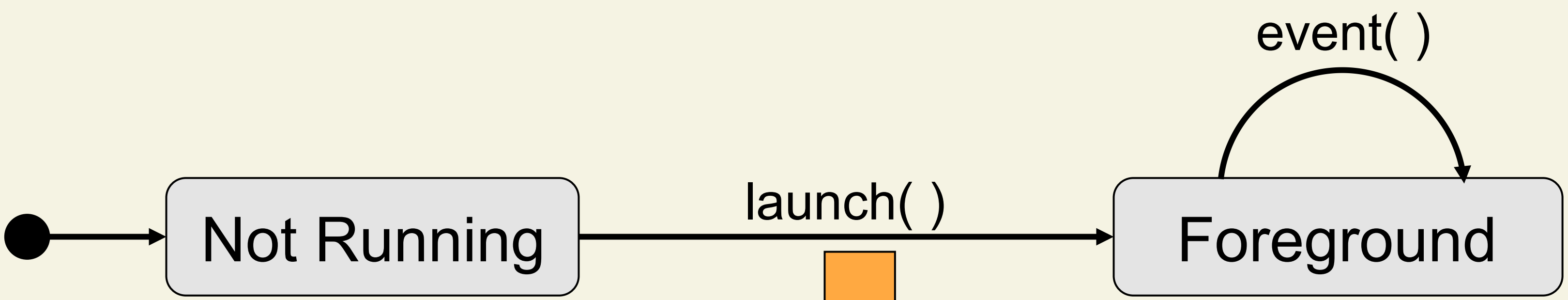
- Advantages
  - Supports many implementation platforms
  - Frees programmers from recurring activities
  - Leads to uniform code
  - Useful to enforce coding conventions (e.g., getters and setters)
  - Models are not mere documentation

Problem: Abstraction Mismatch



- UML models may use different abstractions than the programming language
- Model should not depend on implementation language
- Models cannot always be mapped directly to code

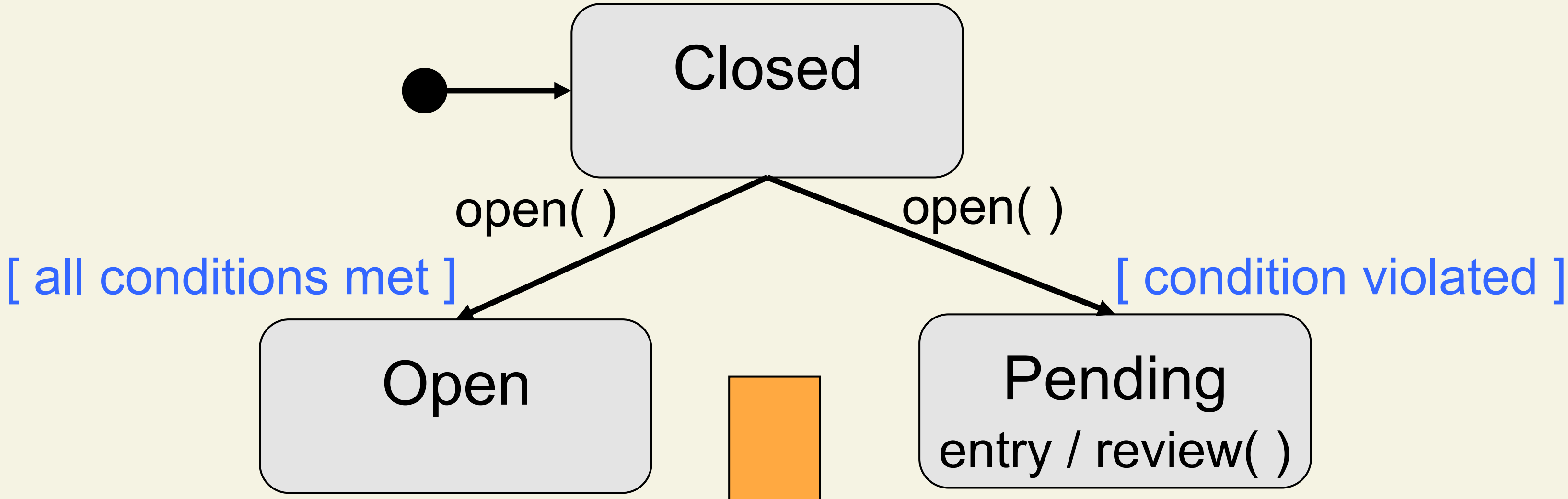
Problem: Specifications are Incomplete




```
class App {
  private State state;
  public App( )
    { state = NOT_RUNNING; }
  public void launch( )
    requires state == NOT_RUNNING;
    { state = FOREGROUND; }
  public void event( )
    requires state == FOREGROU
    { }
}
```

Where is the interesting behavior?

Problem: Specifications may be Informal




```
public void open( )
  requires state == CLOSED;
  requires "all conditions met" || "condition violated";
{
  if ( "all conditions met" )  state = OPEN;
  else { state = PENDING; review( ); }
}
```

How to map informal specifications?

Problem: Switching between Models and Code

- Code has to be changed manually
  - Add interesting behavior
  - Clarify informal specifications
  - Implement incomplete specifications

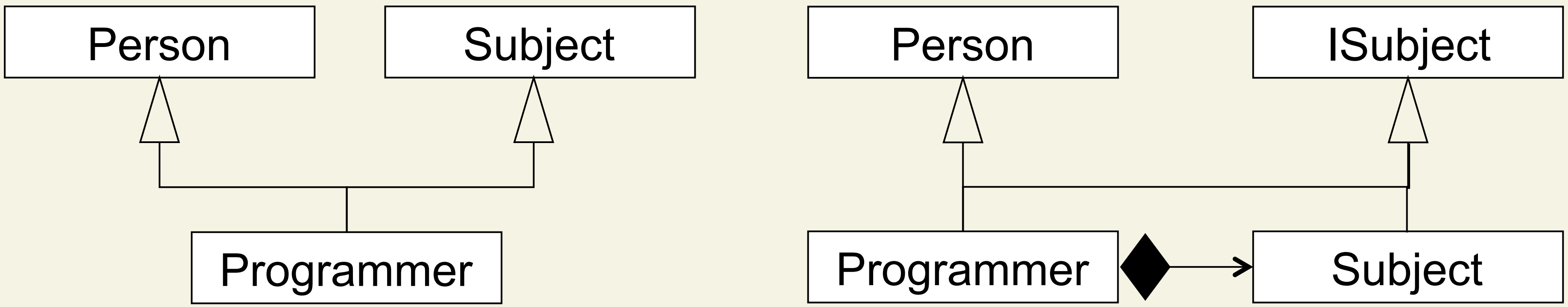
- Modification of code requires complicated synchronization between code and models

Model-Driven Development: Reality

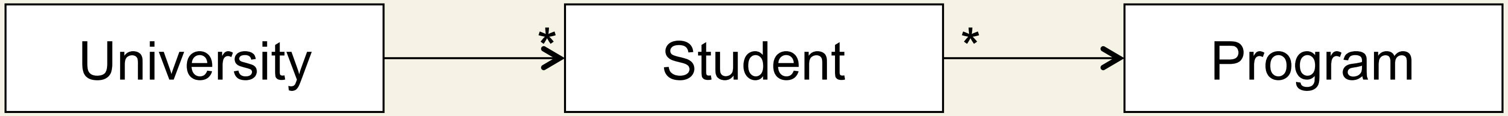
- Works in specific domains (e.g., business process modeling)
- Code generation works for basic properties
- Interesting code is still implemented manually
- Problems
  - Maintaining code that has no models (reverse-engineering)
  - Once code has been modified manually, going back to the model is difficult (or impossible)

Mapping Classes and Inheritance

- Classes may be split into interfaces and implementation classes
- Attributes should be non-public
  - Generate getters and setters with appropriate visibility
- Methods are straightforward
- Inheritance can be mapped to inheritance or subtyping plus aggregation and delegation



Mapping Associations



- Associations are typically mapped to fields

```
class University {
  Set<Student> students;
  … }
```

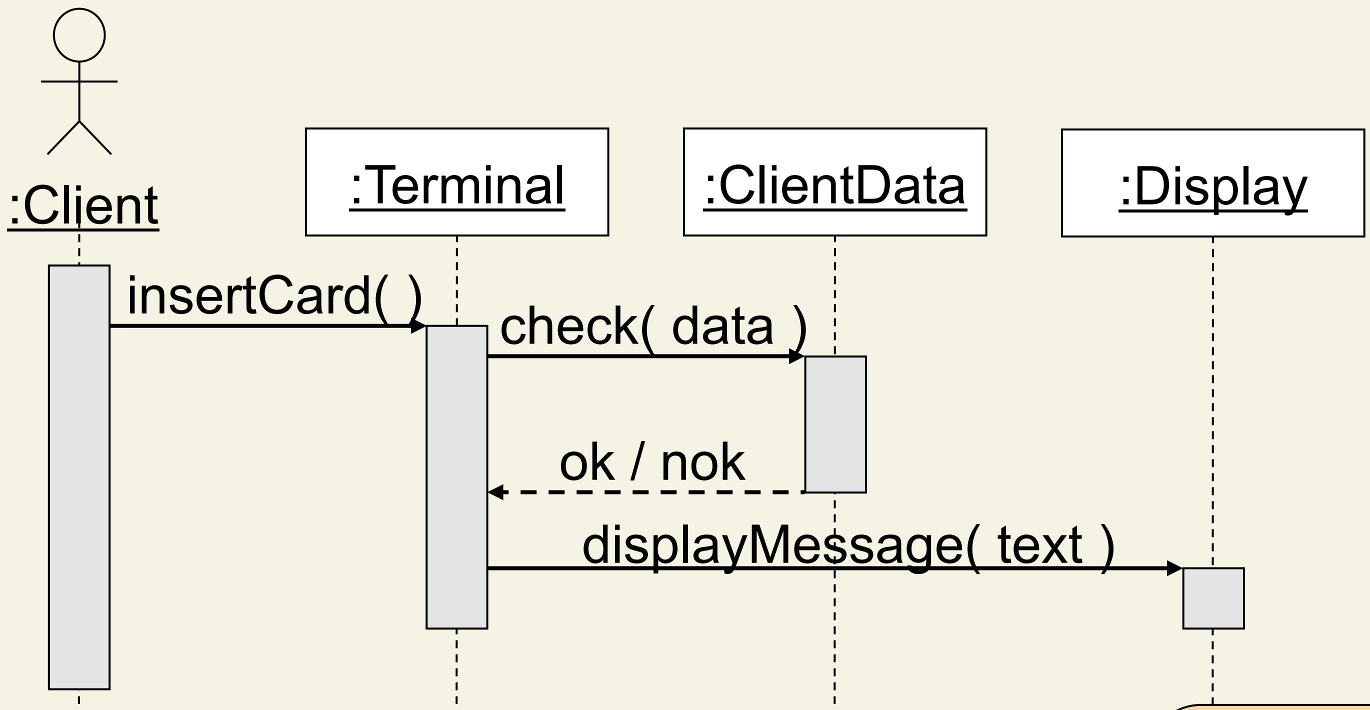
```
class Student {
  Program major;
  … }
```

or separate objects (collections)

```
class University {
  Map<Student, Program> enrollment;
  … }
```

```
class Student {
  … }
```

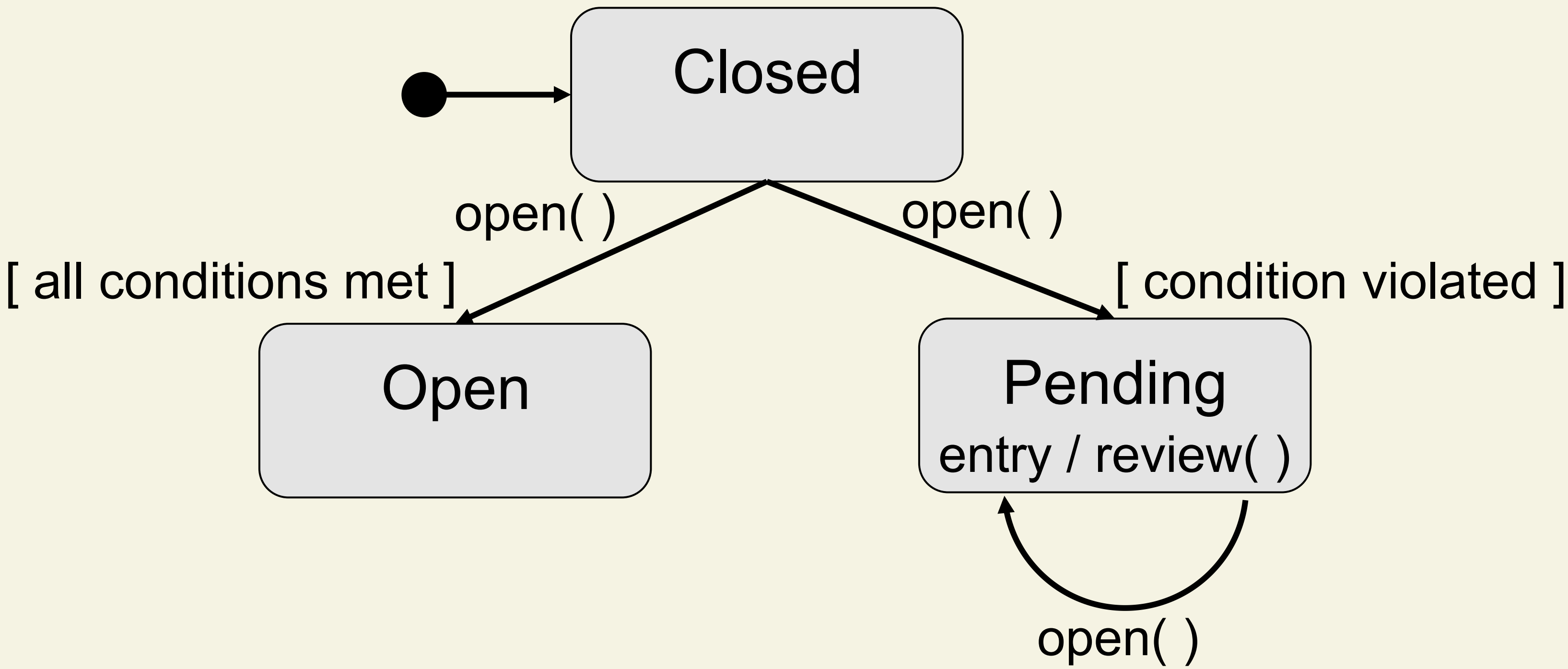
Mapping Sequence Diagrams




```
public void insertCard( ) {
  boolean res = clientData.check( data );
  display.displayMessage( text );
}
```

Synchronous messages are implemented by method calls

Mapping State Diagrams



Mapping State Diagrams (cont'd)

```
public void open( ) throws … {
  switch( state ) {
    case CLOSED:
      if ( "all conditions met" )
        state = OPEN;
      else {
        state = PENDING;
        review( );
      }
      break;
    case PENDING:
      review( );
      break;
    default:
      throw new UnexpectedStateException( );
  }
}
```

Introduce state variable for current state

Check condition of transition

Transition

Perform action

Illegal state or message

Informal Modeling: Summary

Strengths

- Describe particular views on the overall system
- Omit some information or specify it informally
- Graphical notation facilitates communication

Weaknesses

- Precise meaning of models is often unclear
- Incomplete and informal models hamper tool support
- Many details are hard to depict visually

3. Modeling and Specification

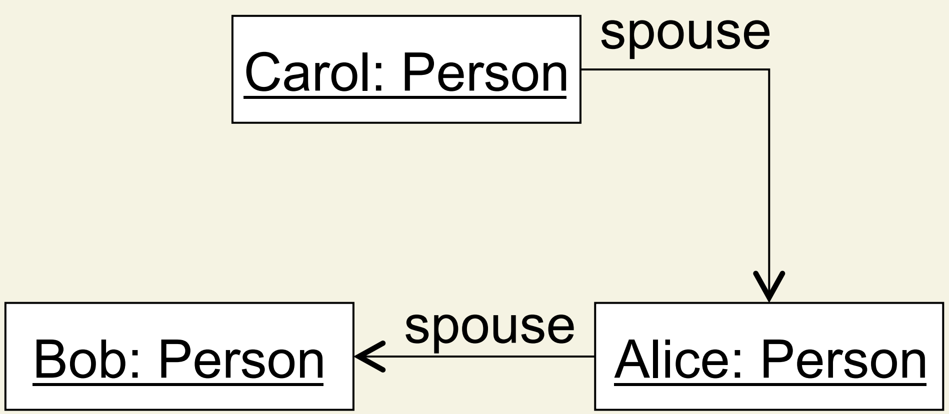
3.1 Source Code

3.2 Informal Models

3.3 Formal Models

Formal Modeling

- Notations and tools are based on mathematics, hence precise
- Typically used to describe some aspect of a system
- Formal models enable automatic analysis
  - Finding ill-formed examples



  - Checking properties

context SavingsAccount **inv**:
self.amount >= 0

Alloy

- Alloy is a formal modeling language based on set theory
- An Alloy model specifies a collection of constraints that describe a set of structures
- The Alloy Analyzer is a solver that takes the constraints of a model and finds structures that satisfy them
  - Generate sample structures
  - Generate counterexamples for invalid properties
  - Visualize structures

Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

Ion Stoica*, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan†
MIT Laboratory for Computer Science
chord@lcs.mit.edu
http://pdos.lcs.mit.edu/chord/

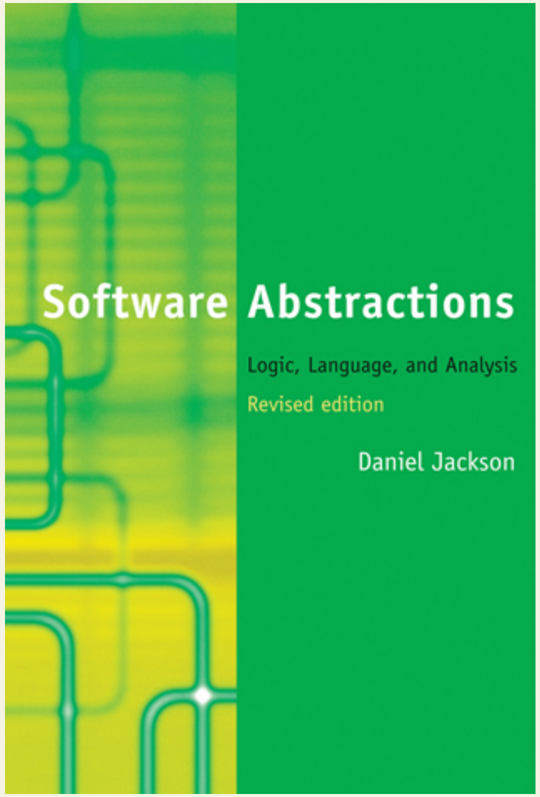
- Chord is a distributed hash table developed at MIT

Three features that distinguish Chord from many other peer-to-peer lookup protocols are its simplicity, provable correctness, and provable performance.

- None of the seven properties claimed invariant of the original version is actually an invariant
- Problems detected through formal modeling

Alloy Documentation and Download

- Documentation
  - Useful tutorials available at alloy.mit.edu
  - Book by Daniel Jackson



- Download
  - Get latest version at alloy.mit.edu/alloy/download.html
  - Requires JRE 6

3. Modeling and Specification

3.1 Source Code
3.2 Informal Models
3.3 Formal Models
    3.3.1 Static Models
    3.3.2 Dynamic Models
    3.3.3 Analyzing Models

Signatures

- A signature declares a set of atoms
  - Think of signatures as classes
  - Think of atoms as immutable objects
  - Different signatures declare disjoint sets

```
sig FSObject { }
```

- Extends-clauses declare subsets relations
  - File and Dir are disjoint subsets of FSObject

```
sig File extends FSObject { }
sig Dir extends FSObject { }
```

Operations on Sets

- Standard set operators
  - \+ (union)
  - & (intersection)
  - \- (difference)
  - **in** (subset)
  - = (equality)
  - \# (cardinality)
  - **none** (empty set)
  - **univ** (universal set)
- Comprehensions

```
sig File extends FSObject { }
sig Dir extends FSObject { }
```

```
#{ f: FSObject | f in File + Dir }
                            >= #Dir
```

```
#( File + Dir ) >= #Dir
```

More on Signatures

- Signature can be abstract
  - Like abstract classes
  - Closed world assumption: the declared set contains exactly the elements of the declared subsets
- Signatures may constrain the cardinalities of the declared sets
  - **one**: singleton set
  - **lone**: singleton or empty set
  - **some**: non-empty set

```
abstract sig FSObject { }
sig File extends FSObject { }
sig Dir extends FSObject { }
```

```
FSObject = File + Dir
```

```
one sig Root
        extends Dir { }
```

Fields

- A field declares a relation on atoms
  - f is a binary relation with domain A and range given by expression e
  - Think of fields as associations
- Range expressions may denote multiplicities
  - **one**: singleton set (default)
  - **lone**: singleton or empty set
  - **some**: non-empty set
  - **set**: any set

```
sig A {
  f: e
}
```

```
abstract sig FSObject {
  parent: lone Dir
}
```

```
sig Dir extends FSObject {
  contents: set FSObject
}
```

Operations on Relations

- Standard operators
  - -> (cross product)
  - . (relational join)
  - ~ (transposition)
  - ^ (transitive closure)
  - * (reflexive, transitive closure)
  - <: (domain restriction)
  - >: (range restriction)
  - ++ (override)
  - **iden** (identity relation)
  - [ ] (box join: e1[e2] = e2.e1)

```
abstract sig FSObject {
  parent: lone Dir
}
```

```
sig Dir extends FSObject {
  contents: set FSObject
}
```

```
one sig Root extends Dir { }
```

```
FSObject in Root.*contents
```

All file system objects are contained in the root directory

Relational Join: Example

- Consider a structure with four FSObject atoms
  - r: Root, d1, d2: Dir, f: File

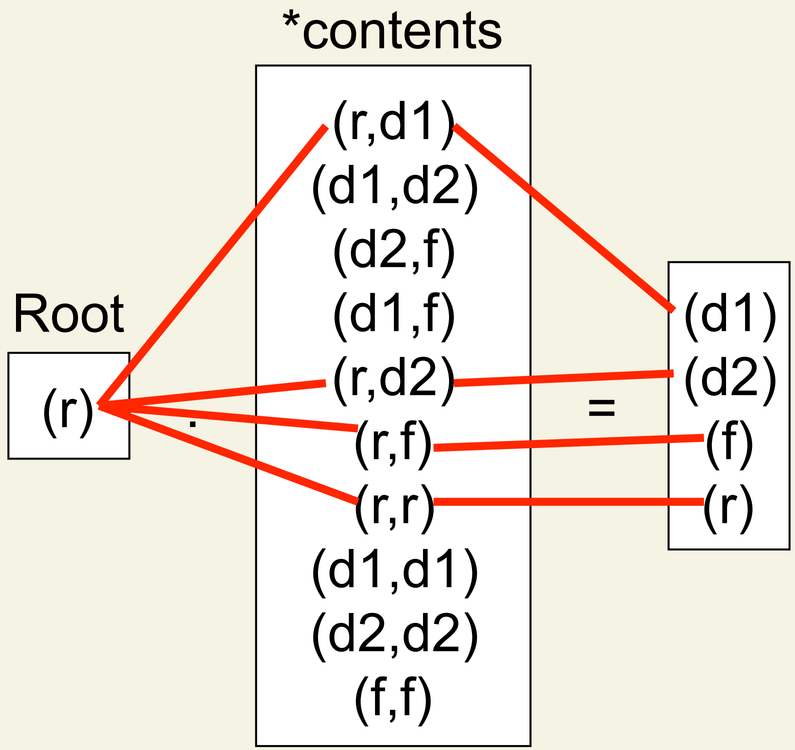
  and contents relation

  (r,d1) (d1,d2) (d2,f)

- The reflexive, transitive closure *contents is

  (r,d1) (d1,d2) (d2,f)
  (d1,f) (r,d2) (r,f)
  (r,r) (d1,d1) (d2,d2) (f,f)

- The relational join Root.*contents is



FSObject **in** Root.*contents

More on Fields

- Fields may range over relations
- Relation declarations may include multiplicities on both sides
  - **one**, **lone**, **some**, **set** (default)

```
sig University {
  enrollment: Student set -> one Program
}
```

- Range expressions may depend on other fields

```
sig University {
  students: set Student,
  enrollment: students set -> one Program
}
```

Constraints

- Boolean operators
  - ! or **not** (negation)
  - && or **and** (conjunction)
  - || or **or** (disjunction)
  - => or **implies** (implication)
  - **else** (alternative)
  - <=> or **iff** (equivalence)
- Four equivalent constraints

F => G **else** H
F **implies** G **else** H
(F && G) || ((!F) && H)
(F **and** G) **or** ((**not** F) **and** H)

- Cardinality constraints
  - **some** e
    e has at least one tuple
  - **no** e
    e has no tuples
  - **lone** e
    e has at most one tuple
  - **one** e
    e has exactly one tuple

no Root.parent

Quantification

- Alloy supports five different quantifiers
  - **all** x: e | F
    F holds for every x in e
  - **some** x: e | F
    F holds for at least one x in e
  - **no** x: e | F
    F holds for no x in e
  - **lone** x: e | F
    F holds for at most one x in e
  - **one** x: e | F
    F holds for exactly one x in e

- Quantifiers may have the following forms
  - **all** x: e | F
  - **all** x: e1, y: e2 | F
  - **all** x, y: e | F
  - **all disj** x, y: e | F

- contents-relation is acyclic

no d: Dir | d **in** d.^contents

Predicates and Functions

- Predicates are named, parameterized formulas

```
pred p[ x1: e1, ..., xn: en ] { F }
```

```
pred isLeave[ f: FSObject ] {
  f in File || no f.contents
}
```

- Functions are named, parameterized expressions

```
fun f[ x1: e1, …, xn: en ]: e { E }
```

```
fun leaves[ f: FSObject ]: set FSObject {
  { x: f.*contents | isLeave[ x ] }
}
```

Exploring the Model

- The Alloy Analyzer can search for structures that satisfy the constraints M in a model

- Find instance of a predicate
  - A solution to
    M &&
    some x1: e1, …, xn: en | F

```
pred p[ x1: e1, ..., xn: en ] { F }
```

```
run p
```

- Find instance of a function
  - A solution to
    M &&
    some x1: e1, …, xn: en,
    res: e | res = E

```
fun f[ x1: e1, …, xn: en ]: e { E }
```

```
run f
```

Exploring the Model: Scopes

- The existence of a structure that satisfies the constraints in a model is in general undecidable

- The Alloy Analyzer searches exhaustively for structures up to a given size
  - The problem becomes finite and, thus, decidable

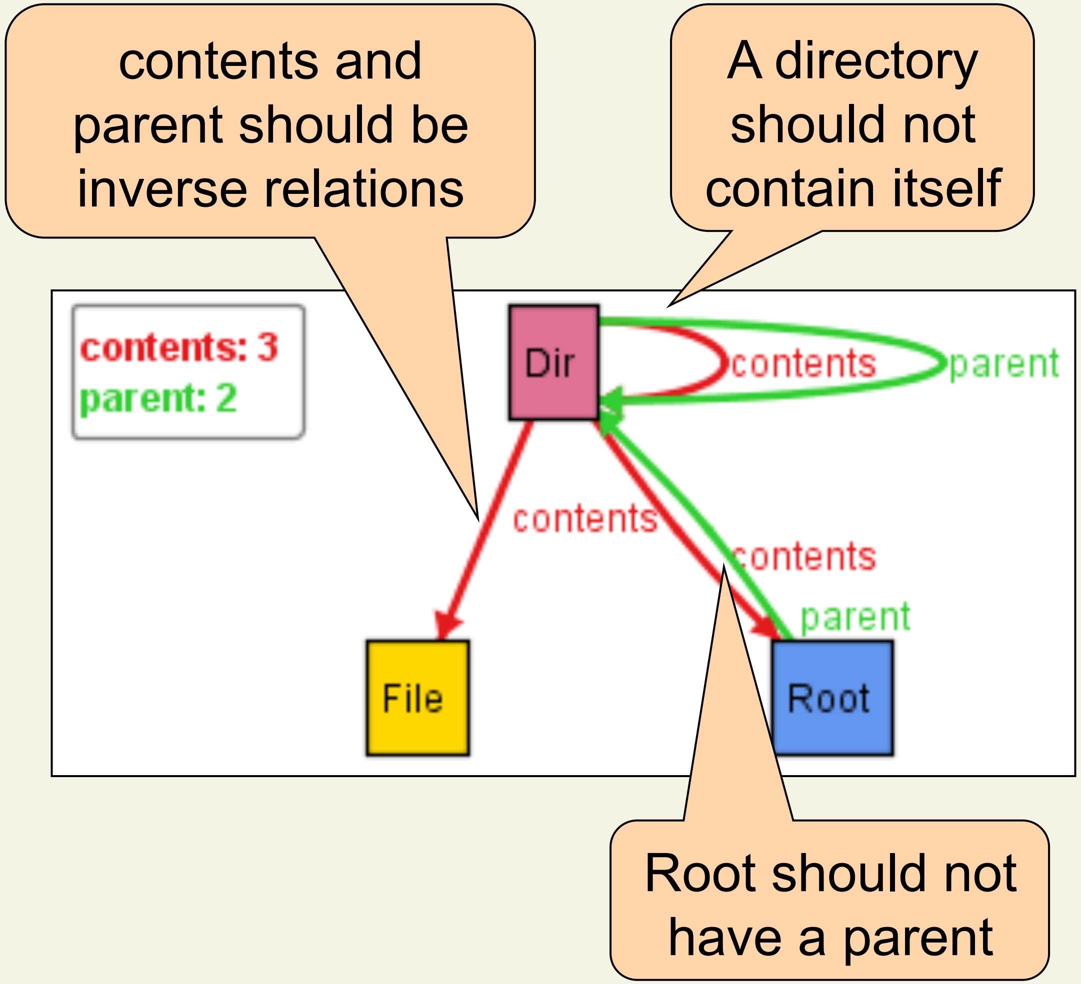
run isLeave
run isLeave **for** 5
run isLeave **for** 5 Dir, 2 File
run isLeave **for exactly** 5 Dir
run isLeave **for** 5 **but** 3 Dir
run isLeave **for** 5 **but exactly** 3 Dir

Exploring the Model: Example

```
abstract sig FSObject {
  parent: lone Dir
}
```

```
sig Dir extends FSObject {
  contents: set FSObject
}
```

```
one sig Root extends Dir { }
```



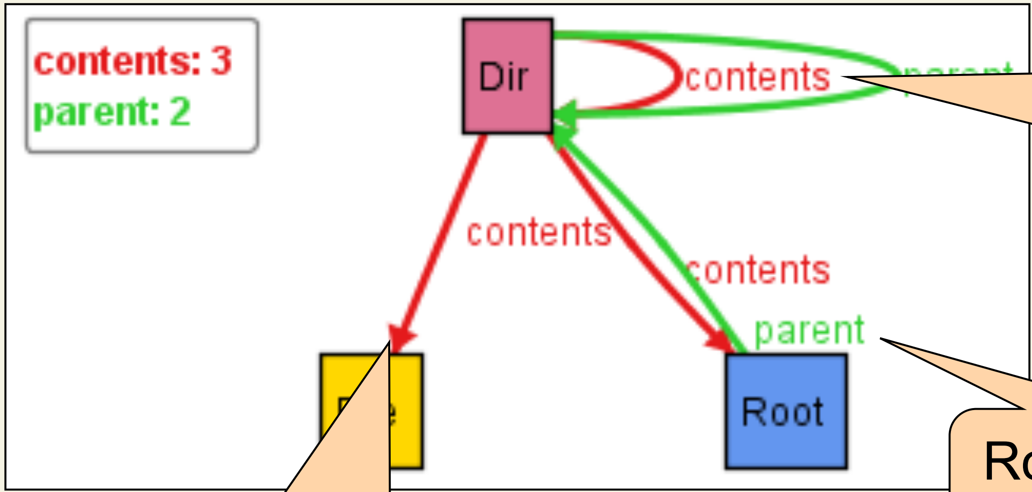
Adding Constraints

- Facts add constraints that always hold
  - **run** searches for solutions that satisfy all constraints

```
fact { F }
fact f { F }
sig S { … } { F }
```

- Facts express value and structural invariants of the model

Adding Constraints: Example



A directory should not contain itself

Root should not have a parent

contents and parent should be inverse relations

fact { **no** Root.parent }

fact { **all** d: Dir, o: d.contents | o.parent = d }

fact { **no** d: Dir | d **in** d.^contents }

Checking the Model

- Exploring models by manually inspecting instances is cumbersome for non-trivial models

- The Alloy Analyzer can search for structures that violate a given property
  - Counterexample to an assertion
  - The search is complete for the given scope

- For a model with constraints M, find a solution to M && !F

assert a { F }

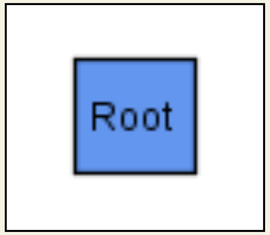
check a *scope*

Checking the Model: Example

- Finding a counterexample

```
pred isLeave[ f: FSObject ] {
  f in File || no f.contents
}
```

```
assert nonEmptyRoot { !isLeave[ Root ] }

check nonEmptyRoot for 3
```



- Proving a property

```
assert acyclic { no d: Dir | d in d.^contents }

check acyclic for 5
```

Validity is checked only within the given scope

```
Executing "Check acyclic for 5"
  Solver=sat4j Bitwidth=0 MaxSeq=0 SkemDepth=1 Symmetry=20
  1047 vars. 63 primary vars. 1758 clauses. 50ms.
  No counterexample found. Assertion may be valid. 33ms.
```

Under and Over-Constrained Models

- Missing or weak facts under-constrain the model
  - They permit undesired structures
  - Under-constrained models are typically easy to detect during model exploration (using **run**) and assertion checking (using **check**)
- Unnecessary facts over-constrain the model
  - They exclude desired structures
- Inconsistencies are an extreme case of over-constraining
  - They preclude the existence of any structure
  - All assertion checks will succeed!

```
fact acyclic {
  no d: Dir | d in d.*contents
}
```

```
assert nonSense { 0 = 1 }
check nonSense ✓
```

Guidelines to Avoid Over-Constraining

- Simulate model to check consistency
  - Use **run** to ensure that structures exist
  - Create predicates with desired configurations and use **run** to ensure they exist

fact acyclic { **no** d: Dir | d **in** d.*contents }

```
pred show { }
run show
```

```
Executing "Run show"
  Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
  0 vars. 0 primary vars. 0 clauses. 3ms.
  No instance found. Predicate may be inconsistent. 0ms.
```

- Prefer assertions over facts
  - When in doubt, check whether current model already ensures a desired property before adding it as a fact

Implementation Documentation: Example

```
class List<E> {
  E[ ] elems;
  int len;
  boolean shared;

  …
}
```

1. elems is non-null
2. When the shared-field is true then the elems-array is immutable
3. When the shared-field is false, the elems-array is used as representation of at most one List object
4. elems is pointed to only by List objects
5. 0 <= len <= elems.length

Reference Counting List: Alloy Model (1)

```
open util/boolean

sig E { }

sig Array {
  length: Int,
  data: { i: Int | 0 <= i && i < length } -> lone E
}
{ 0 <= length }
```

Use library model for booleans

Encode generic type parameter

Introduce array signature to model potential sharing

A fact guaranteed by the Java semantics

Array elements may be null

Reference Counting List: Alloy Model (2)

```
sig List {
  elems: Array,
  len: Int,
  shared: Bool
}
{
  0 <= len && len <= elems.length
}

fact inv3 {
 all disj l1, l2: List | l1.elems = l2.elems => isTrue[l1.shared]
}
```

elems is non-null (inv1)

len is between zero and array size (inv5)

shared conservatively tracks sharing (inv3)

Invariants Revisited

1. elems is non-null
2. When the shared-field is true then the elems-array is immutable
3. When the shared-field is false, the elems-array is used as representation of at most one List object
4. elems is pointed to only by List objects
5. 0 <= len <= elems.length

So far, our model does not contain dynamic behavior

Alloy does not allow the model to constrain fields not declared in the model

Example: Underspecification

```
class University {
  Set<Student> students;
  …
}
```

```
class University {
  Map<Student, Program> enrollment;
  …
}
```

```
class Student {
  Program major;
  …
}
```

```
class Student {
  …
}
```

```
sig Student { }
sig Program { }
sig University { }
sig State {
  enrollment: University -> Student -> one Program
}
```

- The Alloy model leaves the choice of data structure unspecified

Example: Views

```
sig E { }
sig Array {
  length: Int,
  data: { i: Int | 0 <= i && i < length } -> lone E
}
{ 0 <= length }
sig List {
  elems: Array,
  len: Int,
  shared: Bool
}
{ 0 <= len && len <= elems.length }
```

- The Alloy model represents only the structure of the system, not the dynamic behavior
- Some relevant invariants are represented

3. Modeling and Specification

3.1 Source Code

3.2 Informal Models

3.3 Formal Models

- 3.3.1 Static Models
- 3.3.2 Dynamic Models
- 3.3.3 Analyzing Models

Dynamic Behavior

- Alloy has no built-in model of execution
  - No notion of time or mutable state

- State or time have to be modeled explicitly

```
sig Array {
  length: Int,
  data: { i: Int | 0 <= i && i < length } -> lone E
}
```

```
pred update[ a, a': Array, i: Int, e: E ] {
  a'.length = a.length &&
  a'.data = a.data ++ i -> e
}
```

Declarative Specifications

- Alloy specifications are purely declarative
  - The describe what is done, not how it is done
  - Specifications abstract over irrelevant details

```
int find( int[ ] array, int v ) {
  for( int i = 0; i < array.length; i++ )
    if( array[ i ] == v )  return i;
  return -1;
}
```

```
int find( int[ ] array, int v ) {
  if( 256 <= array.length ) {
    // perform parallel search
  } else {
    // sequential search like before
  }
}
```

```
pred find[ a: Array, v: Int, res: Int ] {
  a.data[ res ] = v ||
  res = -1 && (no i: Int | a.data[ i ] = v)
}
```

Describing Mutation via Different Atoms

- Alloy models describe operations declaratively
  - Relating the atoms before and after the operation

```
pred update[ a, a': Array, i: Int, e: E ] {
  a'.length = a.length &&
  a'.data = a.data ++ i -> e
```

A regular identifier

Equality, not assignment

- Modeling mutations via different atoms is cumbersome if atoms occur in several relations

```
pred removeAll[ d, d': Dir ] {
  d'.parent = d.parent &&
  d'.contents = none
}
```

d' is not automatically in d.parent.contents

Abstract Machine Idiom

- Move all relations and operations to a global state

```
sig State { … }
pred init[ s': State, … ] { … }
pred op1[ s, s': State, … ] { … }
pred opn[ s, s': State, … ] { … }
```

- Operations modify the global state

Abstract Machine: Example

```
abstract sig FSObject { }
sig File, Dir extends FSObject { }
```

FileSystem is the global state

```
sig FileSystem {
  live: set FSObject,
  root: Dir & live,
  parent: (live - root) -> one (Dir & live),
  contents: (Dir & live) -> live
}
{
  contents = ~parent
  live in root.*contents
}
```

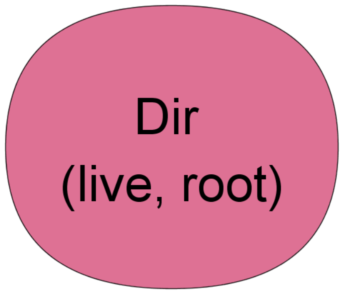
root is a directory in this file system

Every object except root has exactly one parent

Abstract Machine Example: Initialization

```
sig FileSystem {
  live: set FSObject,
  root: Dir & live,
  parent: (live - root) -> one (Dir & live),
  contents: (Dir & live) -> live
}
{
  contents = ~parent
  live in root.*contents
}
```

```
pred init[ s’: FileSystem ] {
  #s’.live = 1
}
```



Abstract Machine Example: Operation

Precondition:
o is a live object other than root

```
pred removeAll[ s, s': FileSystem, o: FSObject ] {
  o in s.live - s.root &&
  s'.live = s.live - o.*(s.contents) &&
  s'.parent = s'.live <: s.parent
}
```

Remove o and everything it (transitively) contains

Restrict domain of parent relation

Abstract Machine Example: Operation (cont'd)

```
sig FileSystem {
  live: set FSObject,
  root: Dir & live,
  parent: (live - root) -> one (Dir & live),
  contents: (Dir & live) -> live
}
{
  contents = ~parent
  live in root.*contents
}
```

Constraints ensure that s.root = s'.root and that s'.contents = ~(s'.parent)

What about s'.root and s'.contents?

```
pred removeAll[ s, s': FileSystem, o: FSObject ] {
  o in s.live - s.root &&
  s'.live = s.live - o.*(s.contents) &&
  s'.parent = s'.live <: s.parent
}
```

In general, we also have to specify what remains unchanged

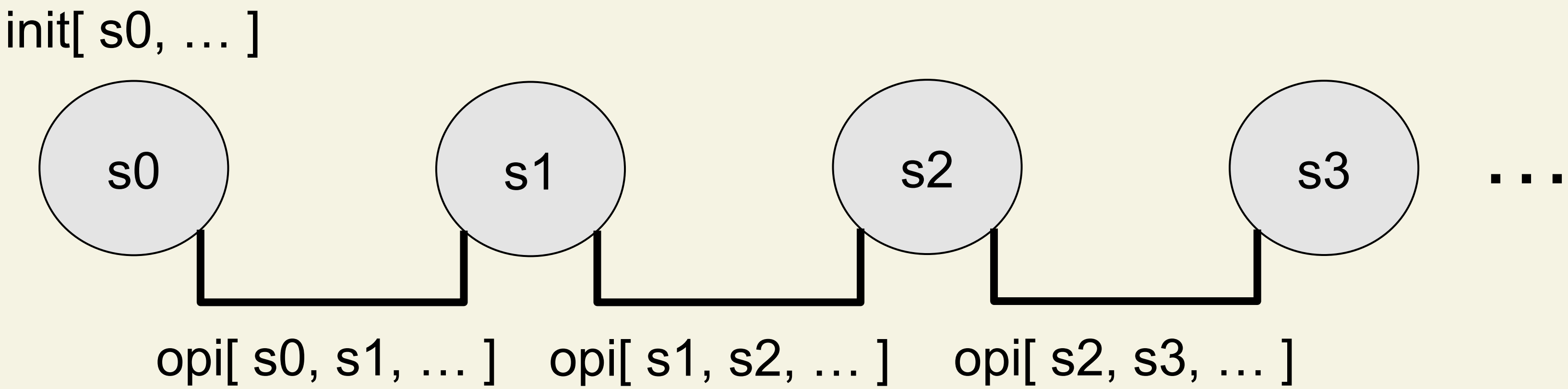
Abstract Machine Executions

sig State { … }

pred init[s’: State, …] { … }

pred op1[s, s’: State, …] { … }

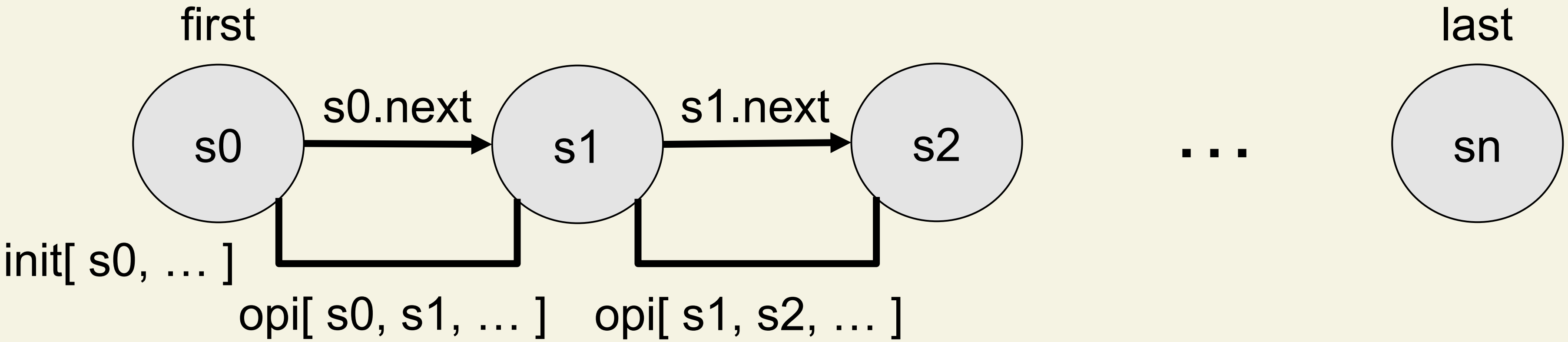
pred opn[s, s’: State, …] { … }



Total Order on States

```
open util/ordering[ State ]
```

Parametric library defines linear order




```
fact execution {
 init[ first, … ] &&
 all s: State - last |
  (some … | op1[ s, s.next, … ])  or
  .
        … | opn[ s, s.next, … ])
```

Initial state is the first in the order

Subsequent states are created by one of the operations

Existential quantifier abstracts over the arguments to the operations

Checking Invariants

- In static models, invariants are expressed as facts
- In dynamic models, invariants can be asserted as properties maintained by the operations

```
sig State { … }
pred init[ s': State, … ] { … }
pred op1[ s, s': State, … ] { … }
pred opn[ s, s': State, … ] { … }
pred inv[ s: State ] { … }
```

```
fact execution {
  init[ first, … ] &&
  all s: State - last |
    (some … | op1[ s, s.next, … ])  or
    …
    (some … | opn[ s, s.next, … ])
}
```

```
assert invHolds {
  all s: State | inv[ s ]
}
```

Checking Invariants: Initialization

```
sig FileSystem {
  live: set FSObject,
  root: Dir & live,
  parent: (live - root) -> one (Dir & live),
  contents: (Dir & live) -> live
}
```

```
pred init[ s': FileSystem ] {
  #s'.live = 1
}
```

```
pred inv[ s: FileSystem ] {
  s.contents = ~(s.parent)
  s.live in s.root.*(s.contents)
}
```

Checking Invariants: Initialization (cont'd)

```
sig FileSystem {
  live: set FSObject,
  root: Dir & live,
  parent: (live - root) -> one (Dir & live),
  contents: (Dir & live) -> live
}
```

```
pred init[ s': FileSystem ] {
  #s'.live = 1 &&
  s'.contents[ s'.root ] = none
}
```

✓

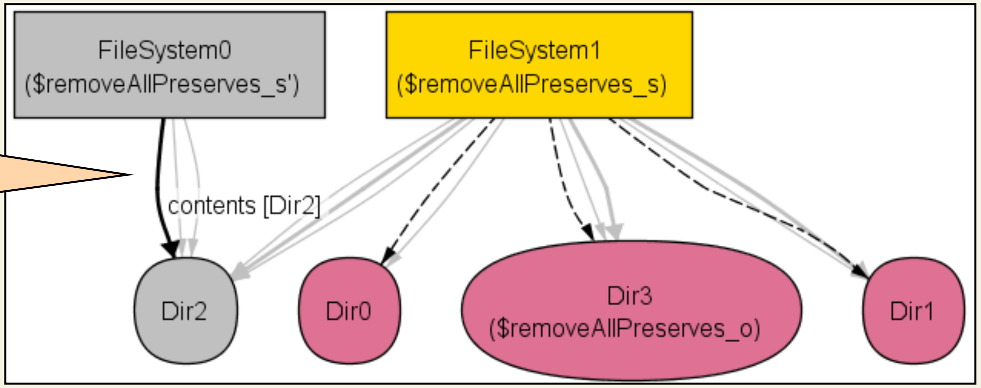
```
pred inv[ s: FileSystem ] {
  s.contents = ~(s.parent)
  s.live in s.root.*(s.contents)
}
```

Checking Invariants: Preservation

```
pred inv[ s: FileSystem ] {
  s.contents = ~(s.parent)
  s.live in s.root.*(s.contents)
}
```

```
pred removeAll[ s, s': FileSystem, o: FSObject ] {
  o in s.live - s.root &&
  s'.live = s.live - o.*(s.contents) &&
  s'.parent = s'.live <: s.parent
}
```

Constraints no longer ensure that s'.contents = ~(s'.parent)



Checking Invariants: Preservation (cont'd)

```
pred inv[ s: FileSystem ] {
  s.contents = ~(s.parent)
  s.live in s.root.*(s.contents)
}
```

```
pred removeAll[ s, s': FileSystem, o: FSObject ] {
  o in s.live - s.root &&
  s'.live = s.live - o.*(s.contents) &&
  s'.parent = s'.live <: s.parent &&
  s'.contents = s.contents :> s'.live
}
```

✓

Temporal Invariants

- The invariants specified and modeled so far were one-state invariants

- Often, one needs to explore or check properties of sequences of states such as temporal invariants

  2. When the shared-field is true then the elems-array is immutable

- Temporal invariants can be expressed
  - Use s.next, lt[s, s'], or lte[s, s'] to relate states

```
pred inv2[ s, s': FileSystem ] {
  s.root = s'.root
}
```

```
assert invtemp {
  all s, s': FileSystem | lte[ s, s' ] => inv2[ s, s' ]
}
```

3. Modeling and Specification

3.1 Source Code

3.2 Informal Models

3.3 Formal Models

- 3.3.1 Static Models
- 3.3.2 Dynamic Models
- 3.3.3 Analyzing Models

Consistency and Validity

- An Alloy model specifies a collection of constraints C that describe a set of structures

- Consistency:
  A formula F is consistent (satisfiable) if it evaluates to true in at least one of these structures

$$\boxed{\exists s \bullet C(s) \land F(s)}$$

- Validity:
  A formula F is valid if it evaluates to true in all of these structures

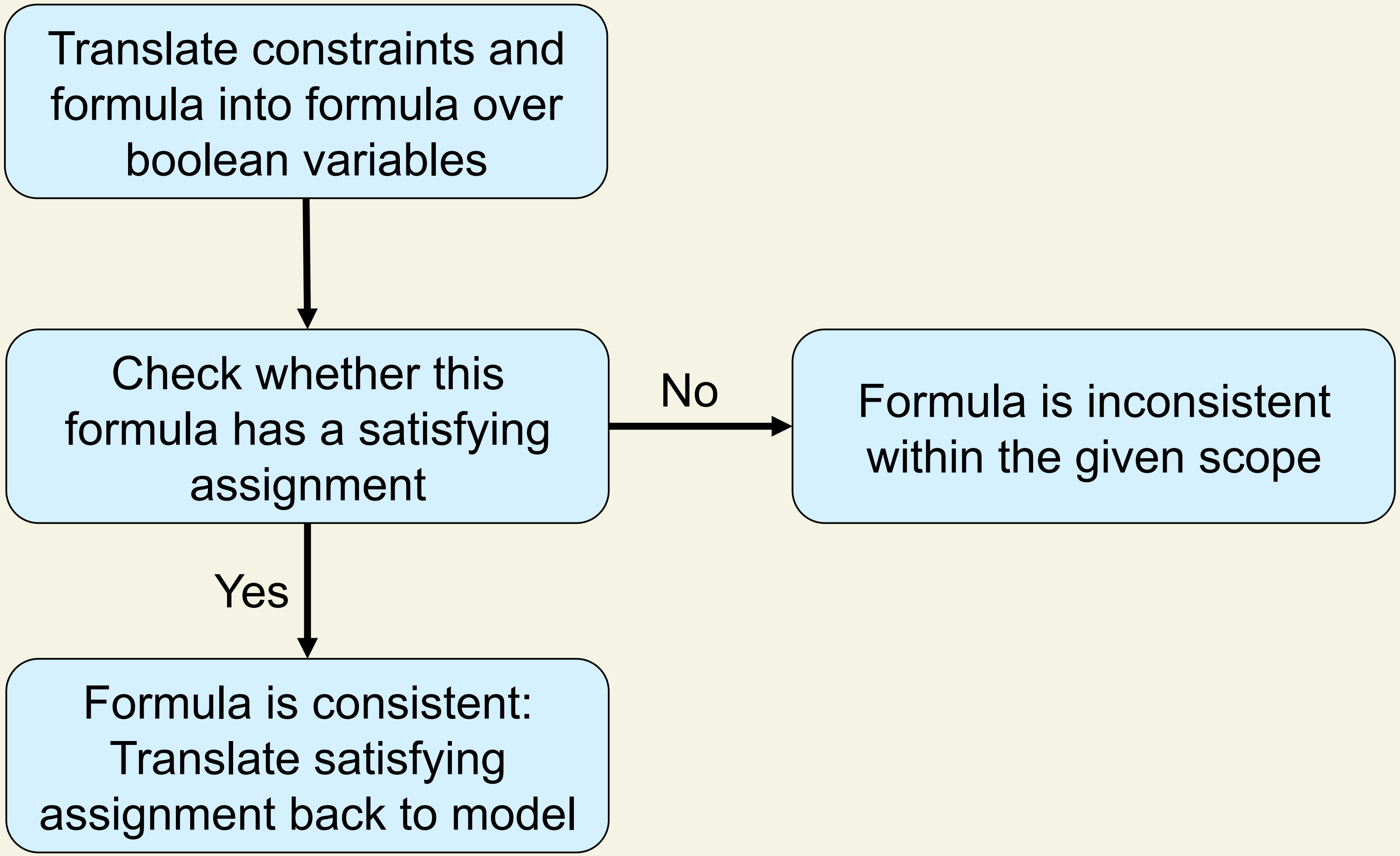
$$\boxed{\forall s \bullet C(s) \Rightarrow F(s)}$$

Analyzing Models within a Scope

- Validity and consistency checking for Alloy is undecidable

- The Alloy analyzer sidesteps this problem by checking validity and consistency within a given scope
  - A scope gives a finite bound on the sizes of the sets in the model (which makes everything else in the model also finite)
  - Naïve algorithm: enumerate all structures of a model within the bounds and check formula for each of them

Consistency Checking



Translation into Formula over Boolean Vars

- Internally, Alloy represents all data types as relations
  - A relation is a set of tuples

```
sig Node {
  next: lone Node
}
```

next is a binary relation in Node × Node

- Constraints and formulas in the model are represented as formulas over relations

```
fact {
  all n: Node | n != n.next
}
```

$\forall n \bullet (n,n) \notin next$

Translation into Boolean Formula (cont'd)

- A relation is translated into boolean variables
  - Introduce one boolean variable for each tuple that is potentially contained in the relation

```
sig Node {
  next: lone Node
}
pred show { }
run show for 3
```

next is a binary relation in Node × Node

$n_{00}, n_{01}, n_{02}, n_{10}, n_{11}, n_{12}, n_{20}, n_{21}, n_{22}$

For the given scope, the next relation may contain nine different tuples

- Constraints and formulas are translated into boolean formulas over these variables

```
fact {
  all n: Node | n != n.next
}
```

$$\neg(n_{00} \wedge n_{01}) \wedge \neg(n_{00} \wedge n_{02}) \wedge \neg(n_{01} \wedge n_{02}) \wedge \neg(n_{10} \wedge n_{11}) \wedge \neg(n_{10} \wedge n_{12}) \wedge \neg(n_{11} \wedge n_{12}) \wedge \neg(n_{20} \wedge n_{21}) \wedge \neg(n_{20} \wedge n_{22}) \wedge \neg(n_{21} \wedge n_{22}) \wedge \neg n_{00} \wedge \neg n_{11} \wedge \neg n_{22}$$

Check for Satisfying Assignments

- Satisfiability of formulas over boolean variables is a well understood problem
  - Find a satisfying assignment if one exists and return UNSAT otherwise
  - The problem is NP-complete
- In practice, SAT solvers are extremely efficient

$$\neg(n_{00} \land n_{01}) \land \neg(n_{00} \land n_{02}) \land \neg(n_{01} \land n_{02}) \land$$
$$\neg(n_{10} \land n_{11}) \land \neg(n_{10} \land n_{12}) \land \neg(n_{11} \land n_{12}) \land$$
$$\neg(n_{20} \land n_{21}) \land \neg(n_{20} \land n_{22}) \land \neg(n_{21} \land n_{22}) \land$$
$$\neg n_{00} \land \neg n_{11} \land \neg n_{22}$$

n	0	1	2
0	F	F	F
1	F	F	T
2	F	T	F

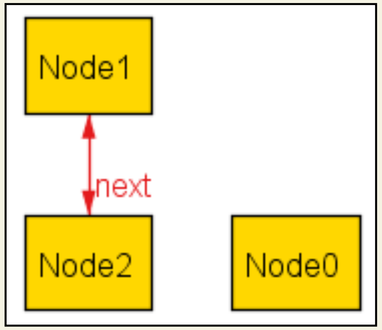
Translation Back to Model

- A satisfying assignment can be translated back to relations

n	0	1	2
0	F	F	F
1	F	F	T
2	F	T	F

next = { (1,2), (2,1) }

and then visualized



Interpretation of UNSAT

- If a boolean formula has no satisfying assignment, the SAT solver returns UNSAT

- The boolean formula encodes an Alloy model within a given scope
  - There are no structures within this scope, but larger structures may exist
  - The model may be, but is not necessarily inconsistent

```
sig Node { next: lone Node }
fact { #Node = 4 }
pred show { }
run show for 3
```

```
Executing "Run show for 3"
  Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
  108 vars. 12 primary vars. 161 clauses. 2ms.
  No instance found. Predicate may be inconsistent. 0ms.
```

Validity and Invalidity Checking

- A formula F is valid if it evaluates to true in all structures that satisfy the constraints C of the model

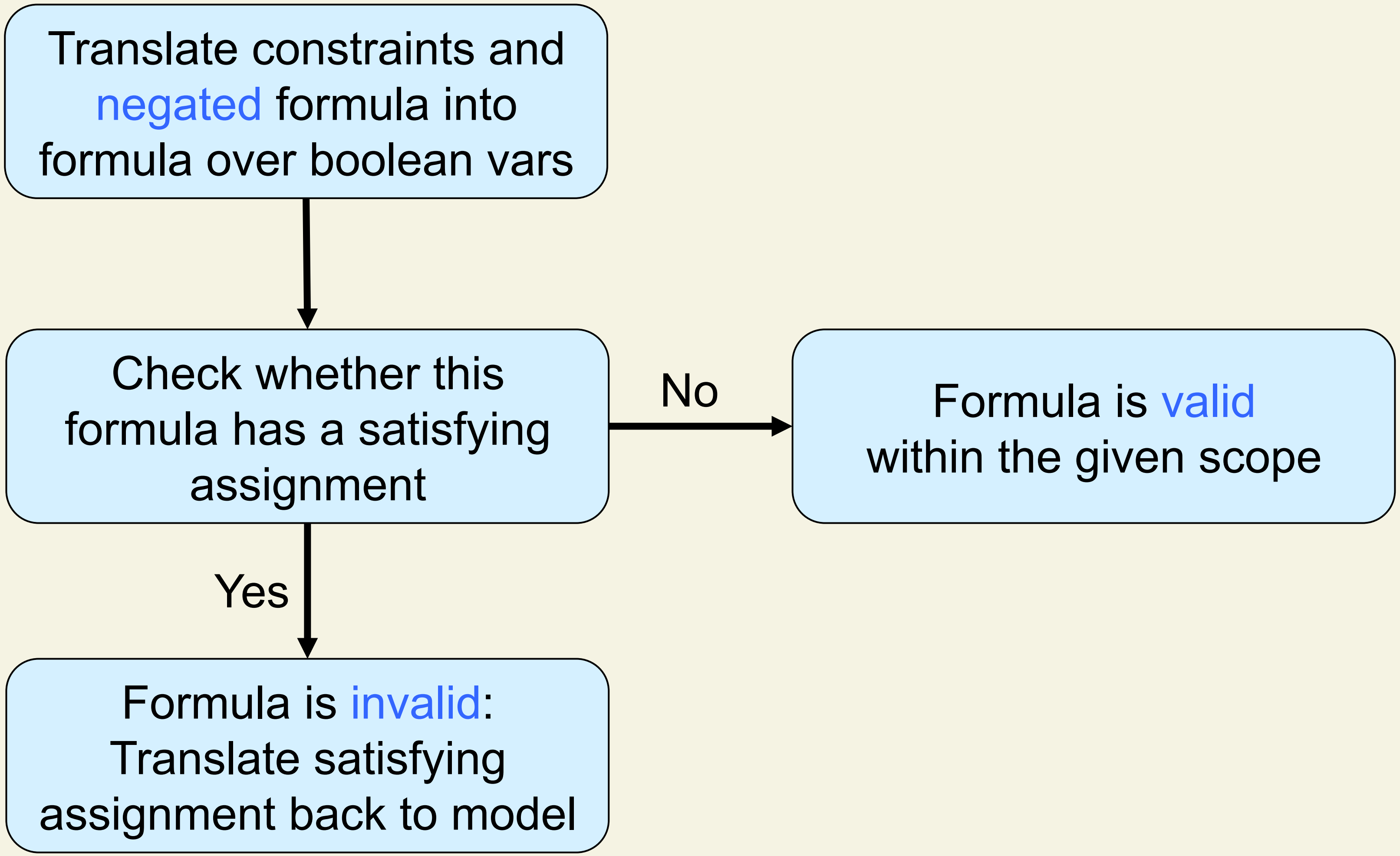
$$\forall s \bullet C(s) \Rightarrow F(s)$$

- Enumerating all structures within a given scope is possible, but would be too slow
- Instead of checking validity, the Alloy Analyzer checks for invalidity, that is, looks for counterexamples

This is a consistency check

$$\neg(\forall s \bullet C(s) \Rightarrow F(s)) \equiv (\exists s \bullet C(s) \wedge \neg F(s))$$

Validity Checking



Interpretation of UNSAT

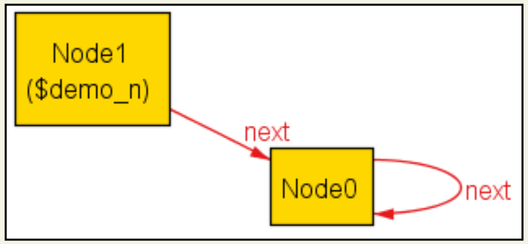
- Validity checking searches for a counterexample within a given scope
  - UNSAT means there are no structures within this scope, but larger structures may exist
  - The model may be, but is not necessarily valid

sig Node { next: Node }
assert demo { **all** n: Node | **some** m: Node | m.next = n }

check demo **for** 1

```
Executing "Check demo for 1"
  Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
  14 vars. 3 primary vars. 18 clauses. 0ms.
  No counterexample found. Assertion may be valid. 0ms.
```

check demo **for** 2



Analyzing Models: Summary

- Consistency checking
  - Performed by **run** command within a scope
  - Positive answers are definite (structures)
- Validity checking
  - Performed by **check** command within a scope
  - Negative answers are definite (counterexamples)

- Small model hypothesis:
  Most interesting errors are found by looking at small instances