CMPE 180-92

Data Structures and Algorithms in C++

December 7 Class Meeting

Department of Computer Engineering
San Jose State University

Spring 2017

Instructor: Ron Mak

www.cs.sjsu.edu/~mak

Unofficial Field Trip

- **Computer History Museum in Mt. View**
  - http://www.computerhistory.org/
  - Provide your own transportation to the museum.
- **Saturday, December 9, 11:30 – closing time**
  - Special free admission (for my students only).
  - Experience a fully restored IBM 1401 mainframe computer from the early 1960s in operation.
  - Do a self-guided tour of the Revolution exhibit.
  - New Make/Software: Change the World exhibit.

The `auto` Keyword

- In a declaration of a variable that is also being initialized, the compiler can <u>infer</u> the type of the variable from the initialization expression.
  - type inference, AKA type determination
- Use `auto` instead of a complicated type name.
  - Examples: Instead of:

```
vector<int>::iterator current = a_container.begin();
map<string, DistanceToCity>::iterator p =
                               cities.lower_bound(new_city.get_name());
```

Use:

```
auto current = a_container.begin();
auto p = cities.lower_bound(new_city.get_name());
```

The `decltype` Pseudo-Function

- Takes a variable as an argument.
- Returns the type associated with the variable.

- Create another variable with the same type.
  - Ensure that two variables have the same type.
  - Example:

```
map<string, int>::iterator start_point;
decltype(start_point) end_point;
```

Function Definitions in Header Files

- If a member function is small, such as a constructor, a getter, or a setter, you can put its definition inside the class declaration.
  - You can have definition code in the header file.
  - Example:

InlineTest1.cpp

```
class InlineTest
{
public:
    InlineTest(int v) : value(v) {}

    int get_value() const { return value; }
    void set_value(const int v) { value = v; }

private:
    int value;
}
```

Function Definitions in Header Files, *cont'd*

- When you put a member function definition inside the class declaration, the compiler can inline the function code.

- Instead of compiling a member function call the usual way, the compiler will instead insert the function code in place of the call.

- This will speed execution (since no calls are executed) but increase the size of the compiled code.

The `inline` Keyword

- If you define a member function outside of the class declaration, you can use the **`inline`** keyword to ask the compiler to inline the function code.
  - The compiler may ignore the keyword.
- It's usually better to let the compiler decide what's best for code optimization.

The `inline` Keyword, *cont*'d

InlineTest2.cpp

```
class InlineTest2
{
public:
    InlineTest2(int v) : value(v) {}

    int get_value() const;
    void set_value(const int v);

private:
    int value;
};

inline int InlineTest2::get_value() const { return value; }
inline void InlineTest2::set_value(const int v) { value = v; }
```

The "Big Three"

- Collectively called the "big three" of a class:
  - overloaded assignment operator
  - copy constructor
  - destructor
- Rule of thumb: If you define a destructor, you should also define a copy constructor and an overloaded assignment operator.
  - Ensure that all three perform in a similar fashion.
  - <u>Do not rely</u> on the default implementations!

The “Big Three”, *cont’d*

- Why does this code crash?

Array1.cpp

```
class Array1
{
public:
    Array1(int s, int v[]);
    ~Array1();

private:
    int size;
    int *vals;
};

Array1::Array1(int s, int v[])
{
    size = s;
    vals = new int[size];
    std::copy(v, v + size, vals);
}
```

The “Big Three”, *cont’d*

- Why does this code crash? *cont’d*

Array1.cpp

```
Array1::~Array1()
{
    delete vals;
    vals = nullptr;
}

int main()
{
    int vals[4] = { 1, 2, 3, 4 };

    Array1 a1(4, vals);
    Array1 a2(a1);

    cout << "Done!" << endl;
    return 0;
}
```

What happens when array **`a2`** goes out of scope?

The “Big Three”, *cont’d*

- Explicitly define a copy constructor:

Array2.cpp

```
class Array2
{
public:
    Array2(int s, int v[]);
    Array2(const Array2& a);
    ~Array2();

private:
    int size;
    int *vals;
};

Array2::Array2(const Array2 &a)
{
    size = a.size;
    vals = new int[a.size];
    std::copy(a.vals, a.vals + size, vals);
}
```

The "Big Three", *cont'd*

Array2.cpp

```
int main()
{
    int vals[4] = { 1, 2, 3, 4 };

    Array2 a1(4, vals);
    Array2 a2(a1);

    a1 = a2;

    cout << "Done!" << endl;
    return 0;
}
```

What happens when
array **`a1`** goes out of scope?

The "Big Three", *cont'd*

- Overload the assignment operator:

Array3.cpp

```
class Array3
{
public:
    Array3(int s, int v[]);
    Array3(const Array3& a);
    ~Array3();

    Array3& operator =(const Array3& a);

private:
    int size;
    int *vals;
};
```

The “Big Three”, *cont’d*

```
Array3& Array3::operator =(const Array3 &a)
{
    if (&a != this)
    {
        size = a.size;
        vals = new int[a.size];
        std::copy(a.vals, a.vals + size, vals);
    }

    return *this;
}
```

Array3.cpp

- This concludes the Big Three!
  - Recently become the Big Five.

References:

- http://www.technical-recipes.com/2011/the-big-three-in-c/
- http://www.cppsamples.com/common-tasks/rule-of-five.html

Lambda Expressions

Person.h

```
#include <string>

using namespace std;

enum class Gender { M, F };

class Person
{
public:
    Person(string f, string l, Gender g);
    virtual ~Person();

    string first;
    string last;
    Gender gender;
};
```

Lambda Expressions, *cont'd*

Person.cpp

```
#include "Person.h"
#include <string>

Person::Person(string f, string l, Gender g)
    : first(f), last(l), gender(g)
{
}

Person::~Person()
{
}
```

Lambda Expressions, *cont'd*

test1.cpp

```
#include <iostream>
#include <vector>
#include "Person.h"

vector<Person> init()
{
    vector<Person> v;
    v.push_back(Person("Ron", "Mak", Gender::M));
    v.push_back(Person("Marie", "Curie", Gender::F));
    v.push_back(Person("Agatha", "Cristie", Gender::F));
    v.push_back(Person("Barack", "Obama", Gender::M));
    return v;
}

ostream& operator <<(ostream& outs, Person &p)
{
    outs << "  {" << "first=" << p.first << ", last=" << p.last
         << ", gender=" << (p.gender == Gender::F ? "F" : "M") << "}";
    return outs;
}
```

Lambda Expressions, *cont'd*

test1.cpp

```
bool is_male(const Person &p)
{
    return p.gender == Gender::M;
}

bool is_C(const Person &p)
{
    return p.last[0] == 'C';
}

vector<Person> match(const vector<Person> people,
                     bool f(const Person &p))
{
    vector<Person> matches;
    for (const Person& p : people) if (f(p)) matches.push_back(p);
    return matches;
}
```

Lambda Expressions, *cont'd*

test1.cpp

```
int main()
{
    vector<Person> people = init();
    vector<Person> males;
    vector<Person> cs;

    cout << "Males:" << endl;
    males = match(people, is_male);
    for (Person& p : males) cout << p << endl;

    cout << endl << "Last name starts with C:" << endl;
    cs = match(people, is_C);
    for (Person& p : cs) cout << p << endl;
}
```

```
Males:
  {first=Ron, last=Mak, gender=M}
  {first=Barack, last=Obama, gender=M}

Last name starts with C:
  {first=Marie, last=Curie, gender=F}
  {first=Agatha, last=Cristie, gender=F}
```

Lambda Expressions, *cont'd*

test2.cpp

```
#include <iostream>
#include <vector>
#include "Person.h"

vector<Person> init()
{
    vector<Person> v;
    v.push_back(Person("Ron", "Mak", Gender::M));
    v.push_back(Person("Marie", "Curie", Gender::F));
    v.push_back(Person("Agatha", "Cristie", Gender::F));
    v.push_back(Person("Barack", "Obama", Gender::M));
    return v;
}

ostream& operator <<(ostream& outs, Person &p)
{
    outs << "  {" << "first=" << p.first << ", last=" << p.last
         << ", gender=" << (p.gender == Gender::F ? "F" : "M") << "}";
    return outs;
}

vector<Person> match(const vector<Person> people, bool f(const Person &p))
{
    vector<Person> matches;
    for (const Person& p : people) if (f(p)) matches.push_back(p);
    return matches;
}
```

Lambda Expressions, *cont'd*

```
int main()
{
    vector<Person> people = init();
    vector<Person> males;
    vector<Person> cs;

    cout << "Males:" << endl;
    males = match(people, [] (const Person &p) -> bool
                          {
                              return p.gender == Gender::M;
                          });
    for (Person& p : males) cout << p << endl;

    cout << endl << "Last name starts with C:" << endl;
    cs = match(people, [] (const Person &p) -> bool
                       {
                           return p.last[0] == 'C';
                       });
    for (Person& p : cs) cout << p << endl;
}
```

```
Males:
   {first=Ron, last=Mak, gender=M}
   {first=Barack, last=Obama, gender=M}

Last name starts with C:
   {first=Marie, last=Curie, gender=F}
   {first=Agatha, last=Cristie, gender=F}
```

test2.cpp

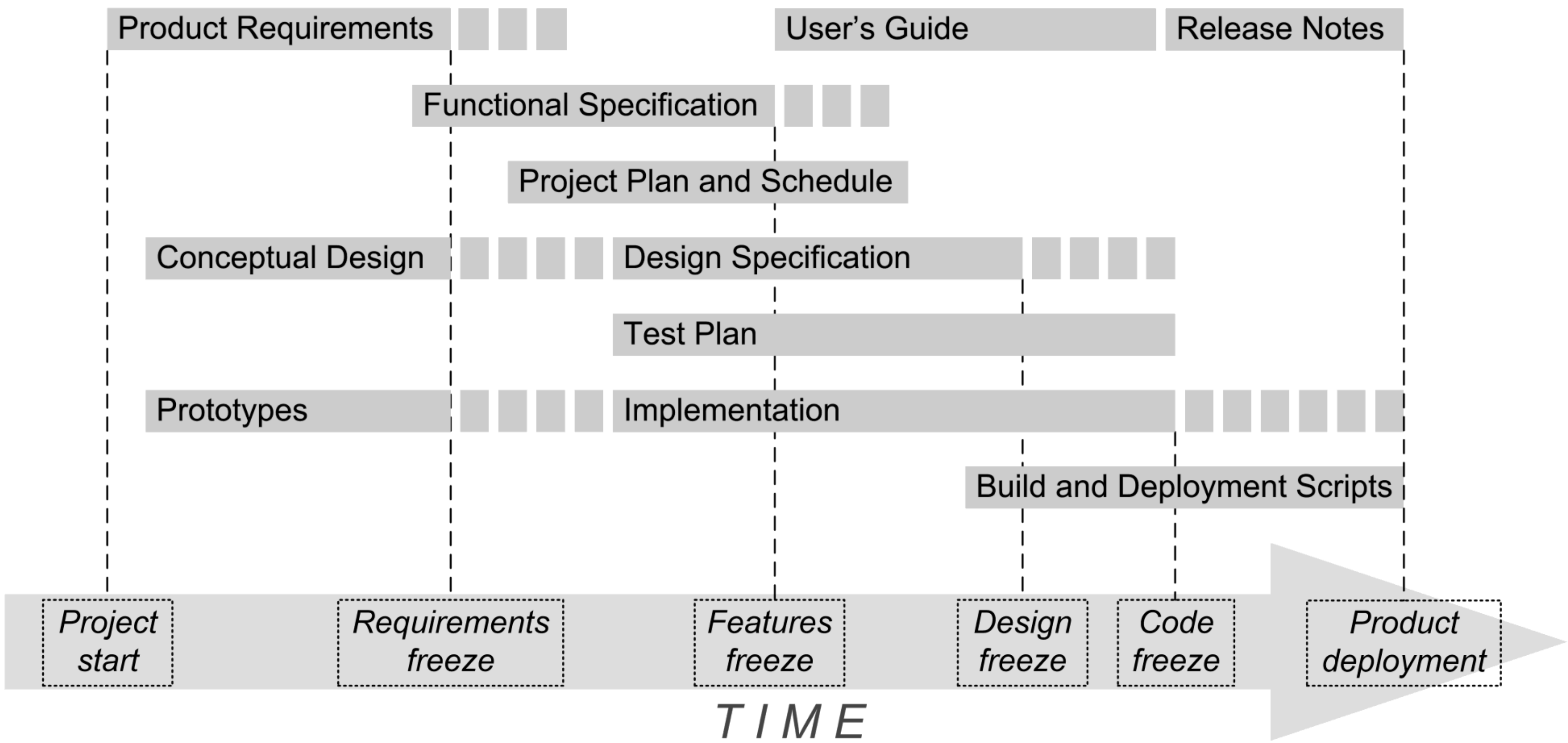
Break

Project Phases

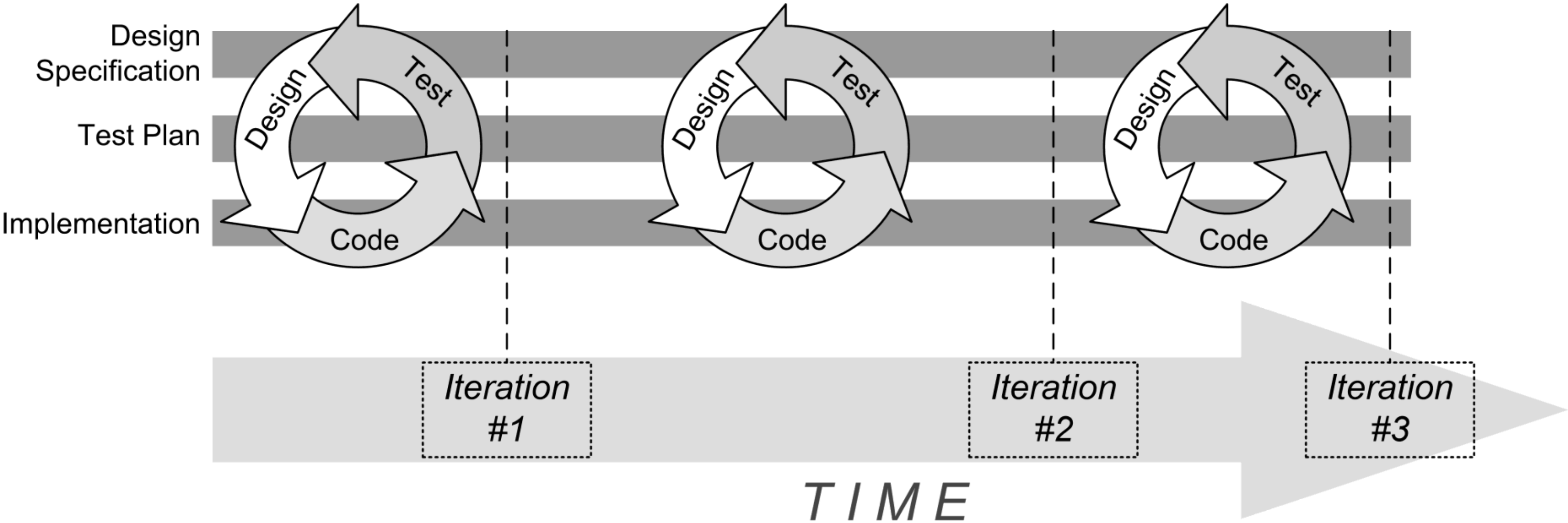
- Requirements elicitation
- Design
- Implementation
- Testing
- Deployment
- Maintenance

- How do we accomplish these phases?

Project Phases, *cont'd*



Project Phases, *cont'd*



- Development is a series of iterations.
- Each iteration is a "mini waterfall" consisting of design, code (implementation), and test.
- Extreme programmers say: design, test, code

The Agile Manifesto for Software Development

We are uncovering better ways of developing software
by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.

Source: http://agilemanifesto.org/

Agile Software Development

- Iterative and incremental development.
- Each iteration consists of:
  - plan (with new requirements)
  - refine design
  - add new code
  - unit and integration testing
- Iterations are short:
  weeks rather than months.
- Iterations are sometimes called "sprints".
  - We do sprints, not marathons!

Agile Software Development

- The initial iteration produces a conceptual design and a prototype.
- Subsequent iterations refine the design and incrementally build the actual product.
- Each subsequent iteration may also include a prototype that is quickly produced (rapid prototyping).
- The initial iteration's prototype and iterative development are the foundation for Rapid Application Development (RAD) tools.

Requirements Elicitation

- Requires communication between the developers and customers.
  - Customer: users, clients, and stakeholders
  - Client: who pays for your application
  - Stakeholder: whoever else is interested in the success of your application (e.g., shareholders)
- Customers can validate the requirements.
- Creates a contract between the customer and the developers.

Requirements Elicitation, *cont'd*

- Result: a Functional Specification written non-technically so that the customers can read and understand it.

Bridging the Gap

- Customers
  - Have a general idea of what the system should do.
  - Have little experience with software development.
  - Are experts in their domain.
- Software developers
  - May have little knowledge of the application domain.
  - Have experience with software technology.
  - Are geeks with poor social skills.

Functional Requirements

- What the system (the application) shall be able to do or allow users to do.
  - *The application <u>shall</u> use GPS to determine the user's location.*
  - *The application <u>must</u> default to the option most frequently chosen by the users.*
  - *The application <u>must</u> allow the user to choose between a text display or a graphics display.*
  - *The user <u>shall</u> be able to make an online withdrawal or deposit.*

Functional Requirements, *cont'd*

- Describe the interactions between the system and its environment, independent of its implementation.

Nonfunctional Requirements

- Usability, reliability, performance, supportability, etc.
  - *The application must respond to user input within 5 seconds.*
  - *The application shall run on the Windows, Mac, and Linux platforms.*
  - *The new GUI must resemble the old GUI.*
  - *Error messages shall be displayed in English and Spanish.*
- Constraints that the system must meet.

Requirements are Strong Statements

- Use strong declarative statements with “shall” and “must”.
  - *The application shall use GPS to determine the user’s location.*
  - *The application must respond to user input within 5 seconds.*

How to Get Requirements

- Interview future users of your application.
- Observe how the users currently work.
  - Can you improve how they currently do things?
  - Can you make them more productive?
- Stated requirements
  - The customer tells you want he or she wants.
- Implied requirements
  - What do you think the customer wants?

How to Get Requirements, *cont'd*

- Customers don't always know what they want.
  - They will know more after you show them a prototype.
  - They will change their minds.
- It's an iterative process!

How to Get Requirements, *cont'd*

- If the developers force the customers to come up with the requirements too soon, they may make something up!
- Such requirements will most likely be wrong or incomplete and lead you astray.

Where Do Classes Come From?

- Textual analysis
  - Look for nouns and verbs in your requirements.
  - Nouns → classes
    - Some nouns are actors.
  - Verbs → functions
  - Class names should be nouns in the singular form, such as **`Product`**, **`Student`**, **`Mailbox`**.

- How will the classes support the behaviors that your requirements describe?

- Focus on concepts, not implementation.

Class Responsibilities

- Responsibilities correspond to verbs in the requirements.
- Each responsibility should be owned by one and only one class.
- Common mistakes:
  - Assigning a responsibility to an inappropriate class.
  - Assigning too many responsibilities to a class.
  - Ideally, each class should have a single primary responsibility.

Class Responsibilities Example

- class `Automobile`
  - `start()`
  - `stop()`
  - `changeTires()`
  - `drive()`
  - `wash()`
  - `displayOilLevel()`
  - `checkOil()`

Too many responsibilities!

A **cohesive** class does
one thing really well and
does not try to be something else.

- class `Automobile`
  - `start()`
  - `stop()`
  - `displayOilLevel()`
- class `Driver`
  - `drive()`
- class `CarWash`
  - `wash()`
- class `Mechanic`
  - `changeTires()`
  - `checkOil()`

Class Relationships: Dependency

- Class `C` depends on class `D`.
  - Some method of `C` manipulates objects of `D`
  - Example: `Mailbox` objects manipulate `Message` objects.
- Dependency is asymmetric.
  - The `Message` class is not aware of the existence of the `Mailbox` class.
  - Therefore, `Message` objects do not depend on `Mailbox` objects.

Class Relationships: Dependency, *cont'd*

- Loose coupling
  - Minimize the number of dependency relationships.
  - An important way for a design to handle change.

Class Relationships: Aggregation

- Class `C` aggregates class `A`.
  - Objects of class `C` contains objects of class `A` over a period of time.
- A special case of dependency.
  - The "has-a" relationship.
  - Example: An `Inventory` object has a list of `Product` objects.

Class Relationships: Aggregation, *cont'd*

- Multiplicity
  - 1:1 – Example: Each **`Person`** object has a single **`StreetAddress`** object.
  - 1:n – Example: Each **`Inventory`** object has an array of multiple **`Product`** objects.

Class Relationships: Inheritance

- Class `C` inherits from class `S`.
  - The "is-a" relationship.
  - All class `C` objects are special cases of class `S` objects.
  - Class `S` is the superclass of class `C`.
  - Class `C` is a subclass of class `S`.
  - An object of class `C` is an object of class `S`.

Class Relationships: Inheritance

- Aggregation: A `Mailbox` object has a `Message` object.
- Inheritance: A `ForwardedMessage` object is a `Message` object.

UML Diagrams

- A picture is worth a thousand words!
- It is much easier to extract information from a graphical notation than reading a textual document.
- Show your design in graphical UML diagrams.
  - UML: Unified Modeling Language

UML Diagrams, *cont'd*

- There are several different types of UML diagrams.
- For now, we'll use:
  - Class diagrams
  - Sequence diagrams
  - State diagrams

UML Class Diagram

- UML: Unified Modeling Language
- A class diagram has three compartments:

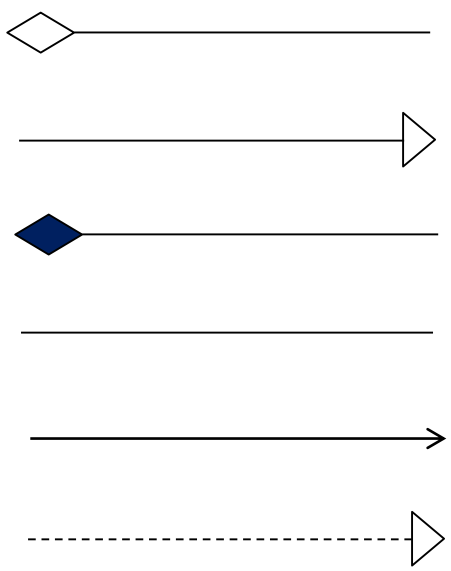
Class Name
Attributes : types
Methods(parms : types) : return type

Example UML Class Diagram

Mailbox
newMessages : vector<Message> savedMessages : vector<Message>
add(msg : Message) : boolean getCurrentMessage() : Message

UML Class Diagram: Relationships

- Relationships among classes using arrows.



Dependency

Aggregation

Inheritance

Composition

Association

Direct association

Interface implementation

UML Class Diagram: Multiplicities

- Multiplicity in a “has” relationship.

Sign	Purpose
*	Zero or more
1..*	One or more
0..1	Zero or one
1	Exactly one

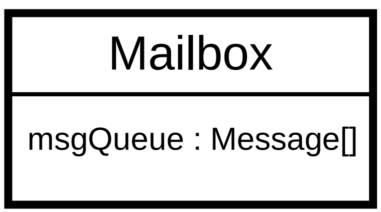
UML Class Diagrams: Association

- A relationship between class `A` and class `B` that lasts as long as class `A` objects and class `B` objects live at runtime.

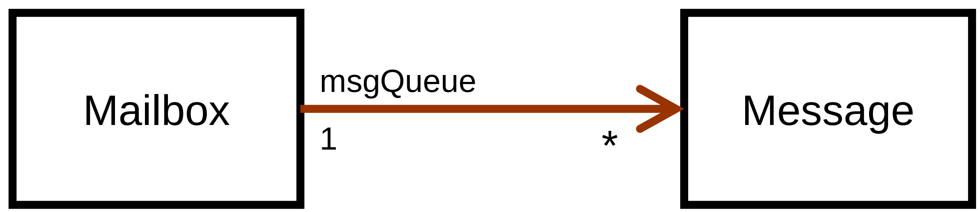
- In general, class `A` has an attribute (field) that is class `B`.

UML Class Diagrams: Association, *cont'd*

- In UML class diagrams, draw a <u>solid line</u> with an <u>open arrowhead</u> from class **A** to class **B**.
- Label the line with the name of the attribute.
  - Don't repeat the attribute inside the class box.
- Can also be an aggregation or a composition.
- Optionally indicate multiplicity.

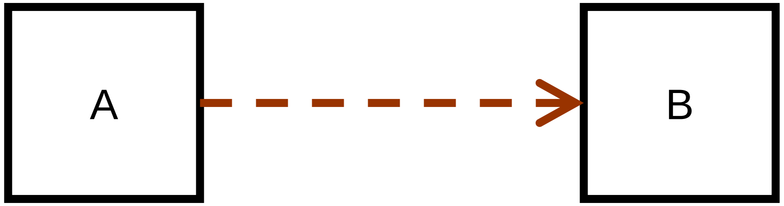


Replace the attribute with the association.



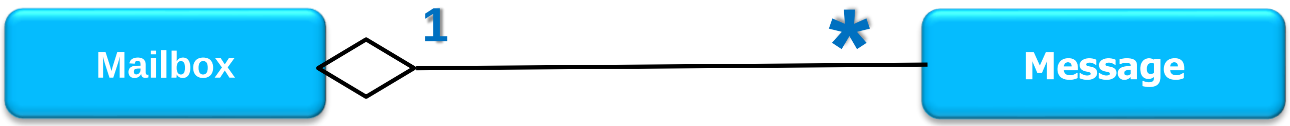
UML Class Diagram: Dependency

- Class **`A`** has a <u>dependency relationship</u> with Class **`B`**, generally a <u>transient</u> relationship.
  - Example: A method of class **`A`** is passed a <u>parameter</u> of class **`B`**.
  - Example: A method of class **`A`** <u>returns a value</u> of class **`B`**.
- In UML diagrams, draw a <u>dashed line</u> with an <u>open arrowhead</u> from class **`A`** to class **`B`**.



UML Class Diagram: Aggregation

- A "has a" relationship.
  - The contained object can have an existence independent of its container.
  - Example
    - A mailbox has a set of messages.
    - A message can exist without a mailbox.
    - Therefore, a mailbox aggregates messages.
- Draw an open diamond at the owner end.

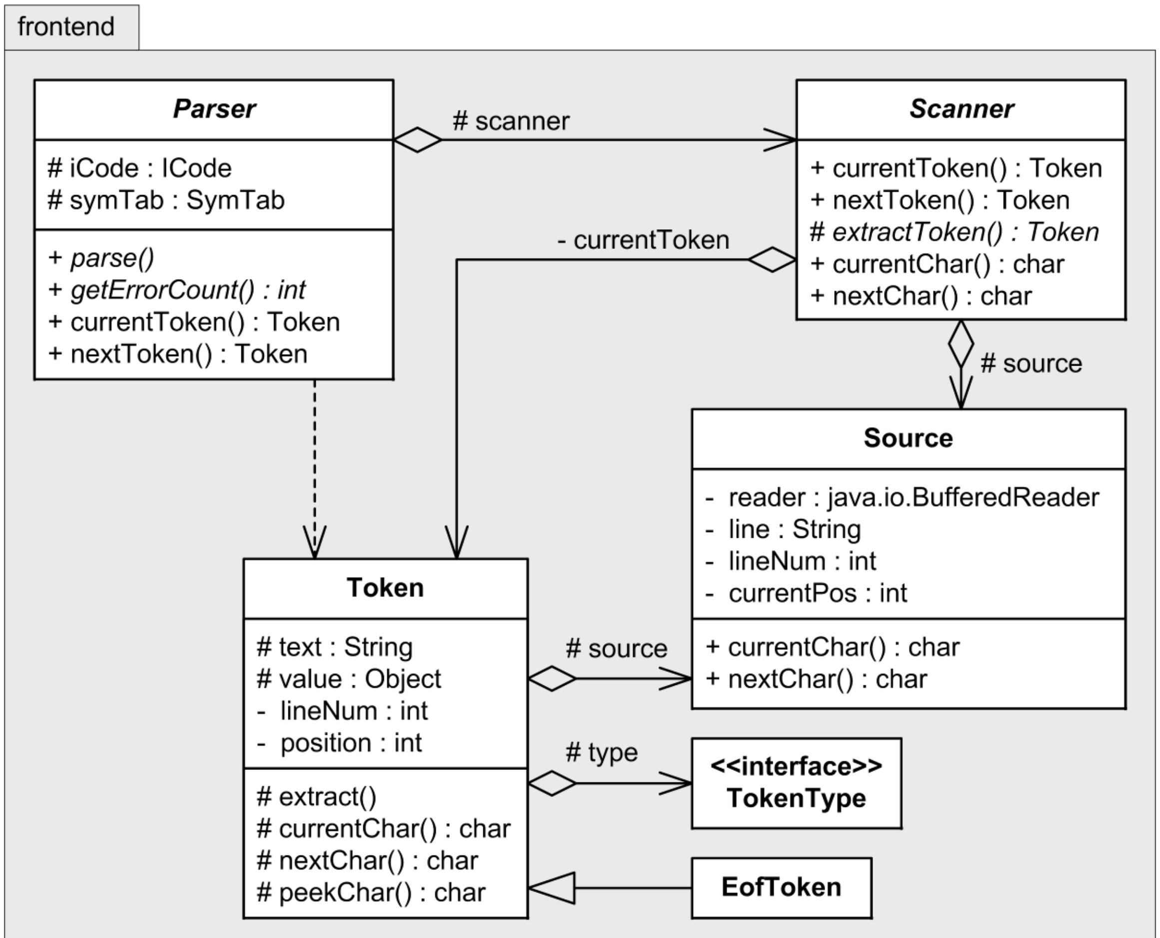


UML Class Diagram: Composition

- Another "has a" relationship.
  - The contained object <u>cannot</u> (logically) have an existence independent of its container.
  - Example
    - A mailbox <u>has a</u> message queue.
    - The message queue <u>cannot</u> (logically) exist without a mailbox.
    - Therefore, a mailbox <u>composes</u> a message queue.
- Draw an <u>filled diamond</u> at the <u>owner end</u>.

Class Diagram Examples

- What's in the `frontend` package of a



UML package diagram

What information can you learn from the class diagrams?

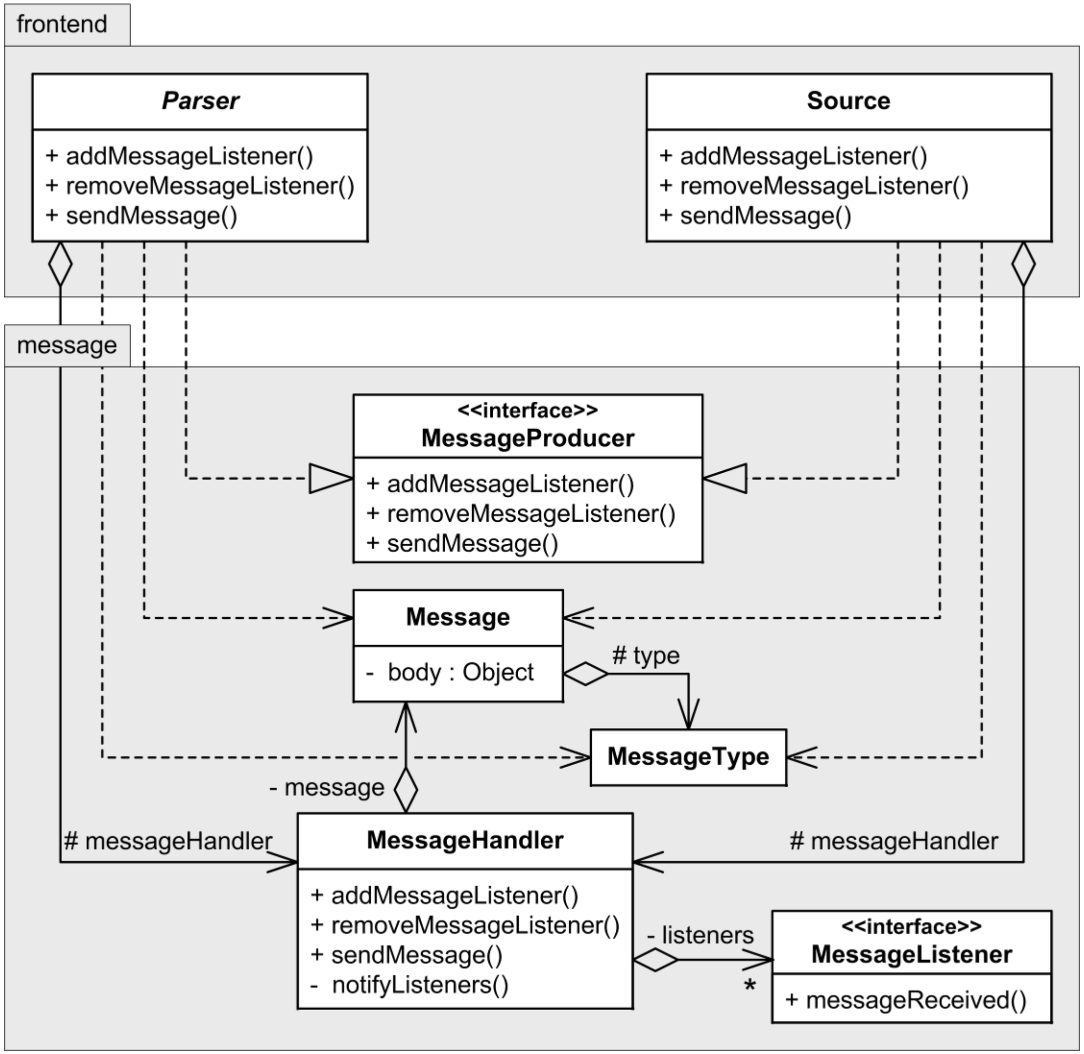
Access control

- \+ public
- – private
- \# protected
- ~ package

From: **Writing Compilers and Interpreters, 3rd ed.**, John Wiley & Sons, 2009.

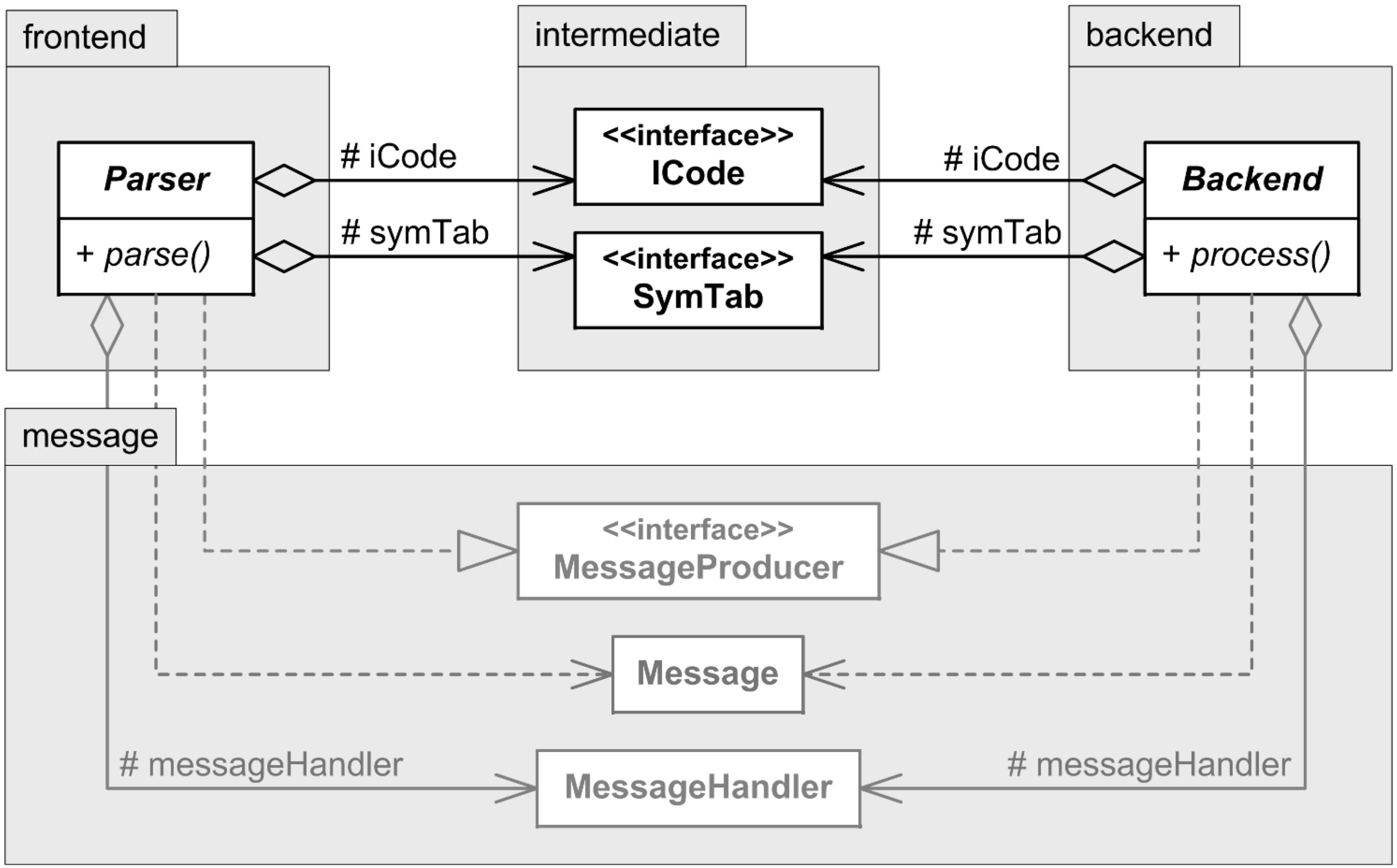
Class Diagram Examples, *cont'd*

- Message handling in the front end of a compiler.
  - **`frontend`** and **`message`** packages



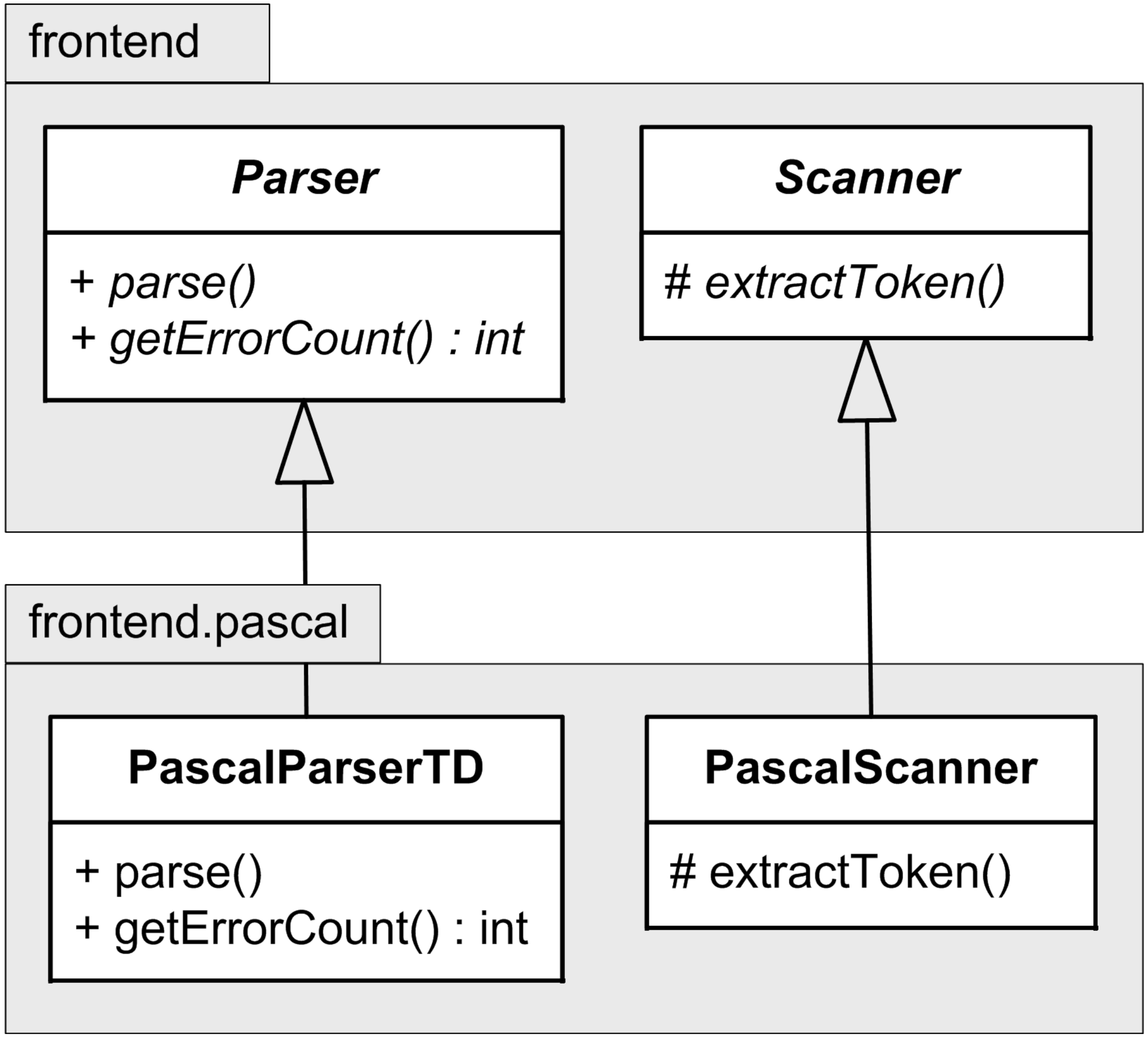
Class Diagram Examples, *cont'd*

- The **`frontend`**, **`intermediate`**, and **`backend`** packages.



Class Diagram Examples, *cont'd*

- Implement the abstract base classes `Parser` and `Scanner` with language-specific subclasses.



Class Diagram Examples, *cont'd*

- The back end can be a code generator or an executor.