Component-based programming

A new programming paradigm

Karel Crombecq

Outline

- History of programming paradigms
- Issues with object-oriented programming
- Component-based programming: an alternative
- Practical issues
  - Initialization
  - Communication
  - Change of mindset
- Implementation: the Cistron library
- Conclusion

Outline

- <u>History of programming paradigms</u>
- Issues with object-oriented programming
- Component-based programming: an alternative
- Practical issues
  - Initialization
  - Communication
  - Change of mindset
- Implementation: the Cistron library
- Conclusion

Definition

- Programming paradigm:
  A **programming paradigm** is a fundamental style of computer programming. Paradigms differ in the concepts and abstractions used to represent the elements of a program (such as objects, functions, variables, constraints, etc.) and the steps that compose a computation (assignment, evaluation, continuations, data flows, etc.).

History (1)

1. Low-level programming languages (assembly & machine code)
   - No abstraction
   - No encapsulation

History (2)

2. Procedural programming languages
   (C, BASIC, Fortran, …)
   - Abstraction using routines/procedures/functions
   - No encapsulation: data and methods are not encapsulated in one package

History (3)

3. Object-oriented programming languages (C++, C#, PHP5, …)
    - Abstraction through classes
    - Encapsulation through private members members
    - Inheritance/polymorphism

History (4)

4. Alternative paradigms:
    - Logic programming (Prolog)
    - Functional programming (Lisp)
    - Component-based programming

What this is NOT about…

- Component-based programming, according to Wikipedia:

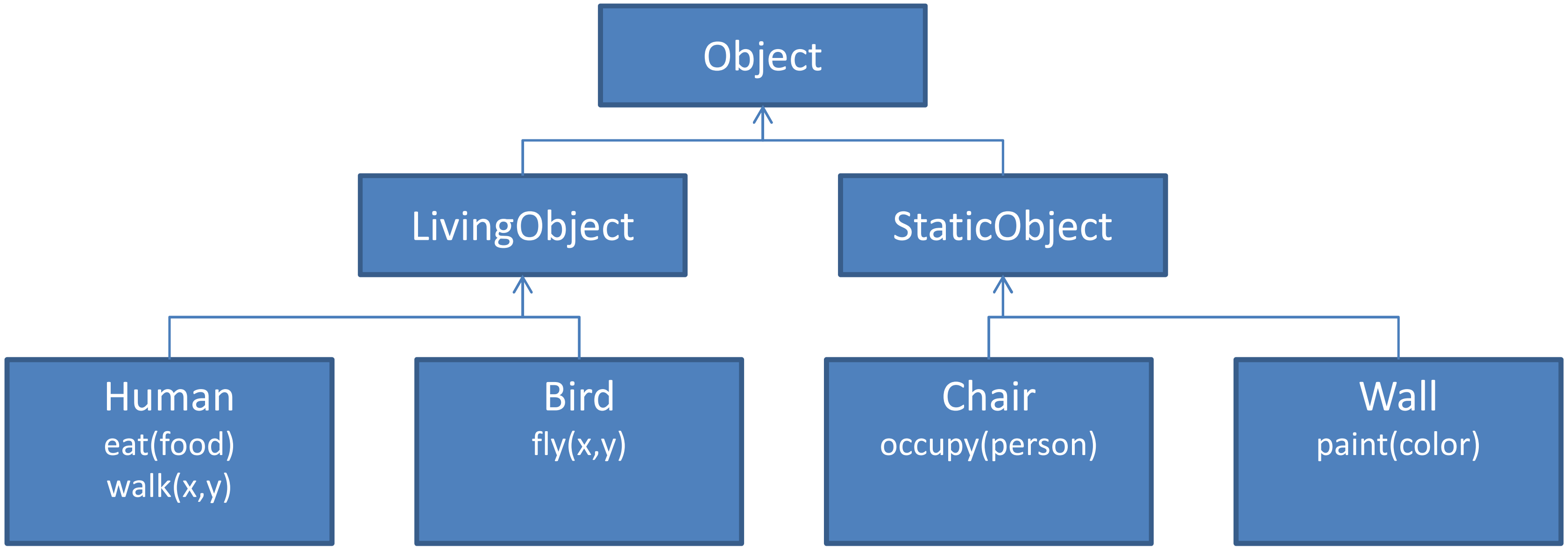
  Component-based software engineering is a branch of software engineering which emphasizes the separation of concerns in respect of the wide-ranging functionality available throughout a given software system.
  An individual **component** is a software package, a web service, or a module that encapsulates a set of related functions (or data).

Outline

- History of programming paradigms
- <u>Issues with object-oriented programming</u>
- Component-based programming: an alternative
- Practical issues
  - Initialization
  - Communication
  - Change of mindset
- Implementation: the Cistron library
- Conclusion

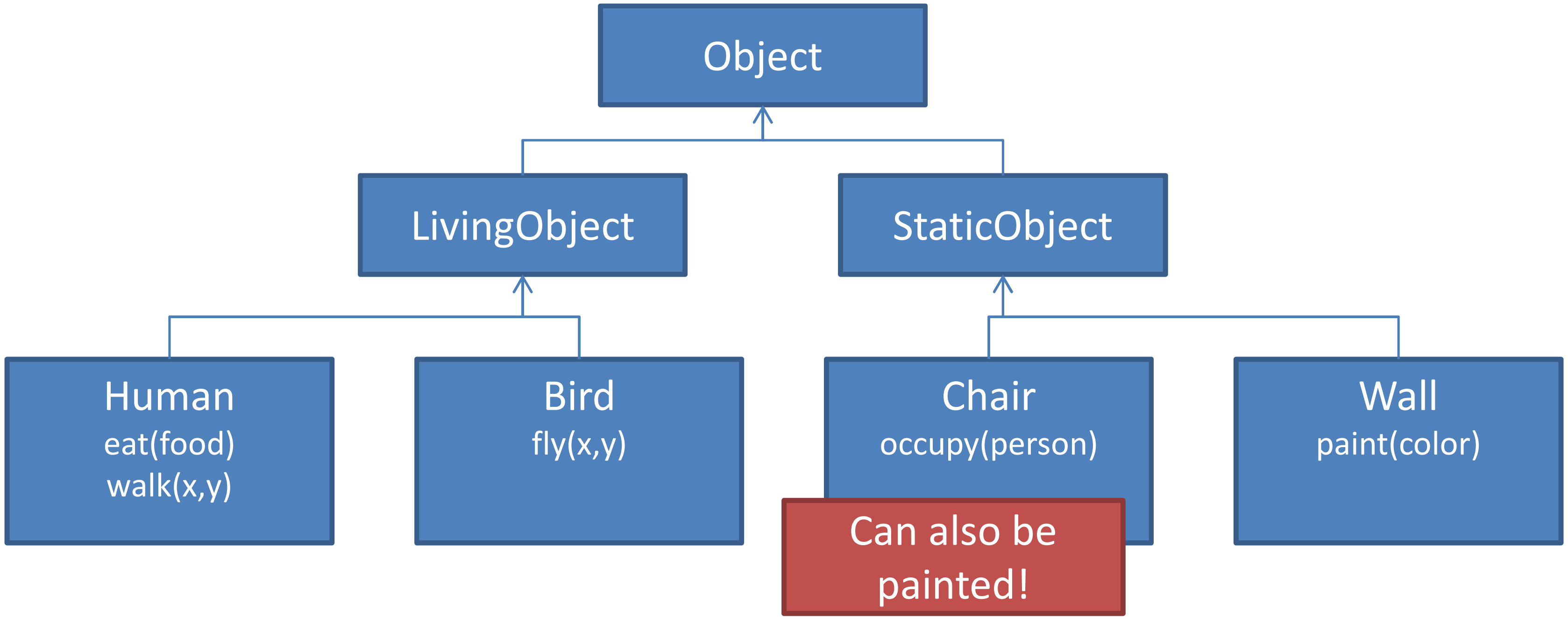
The Blob (anti)pattern

- Modelling our world with traditional OO methodology:



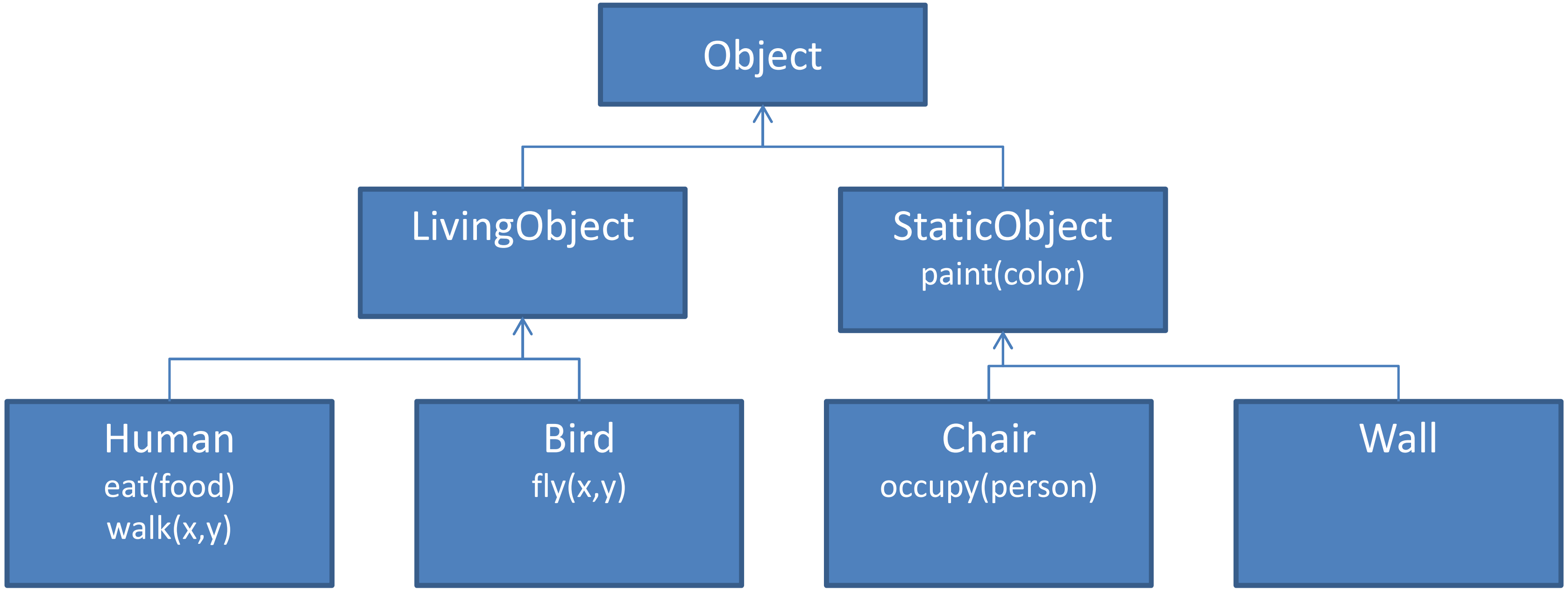
The Blob (anti)pattern

- Modelling our world with traditional OO methodology:



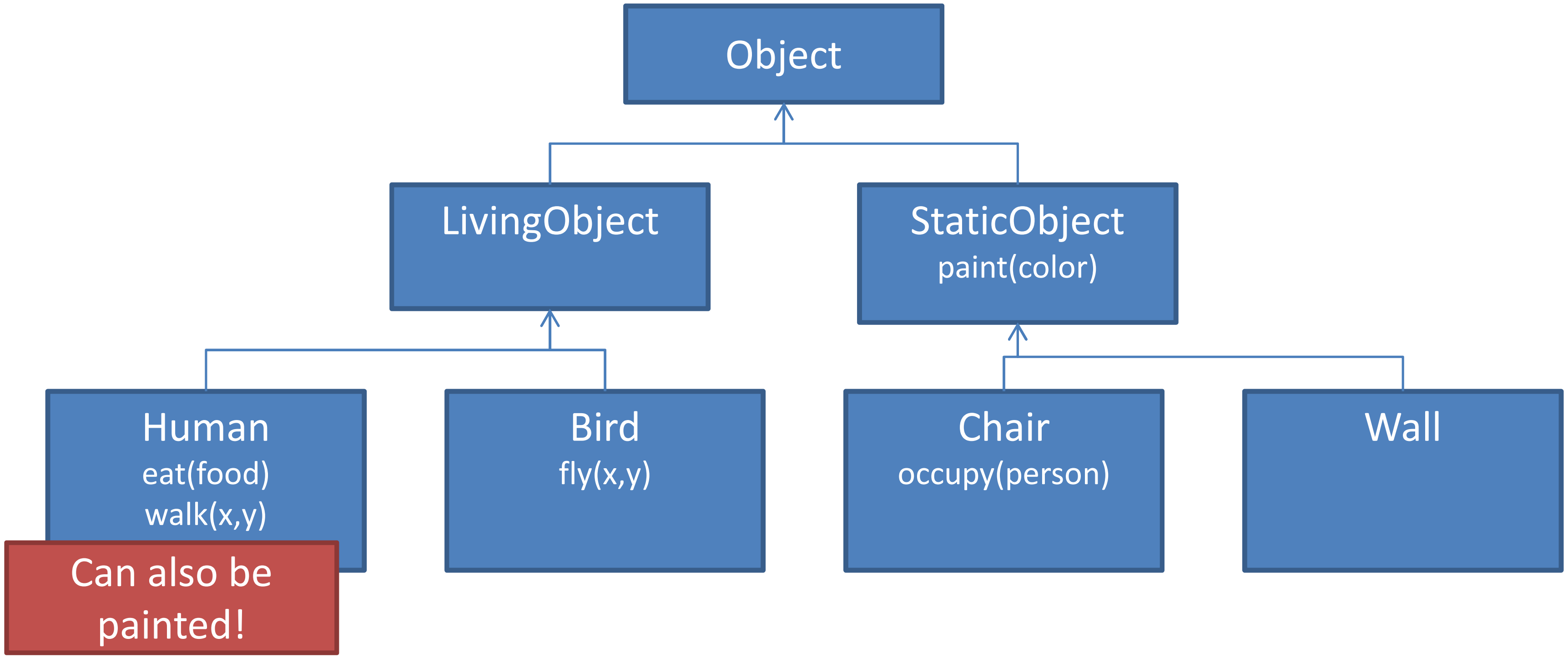
The Blob (anti)pattern

- Modelling our world with traditional OO methodology:



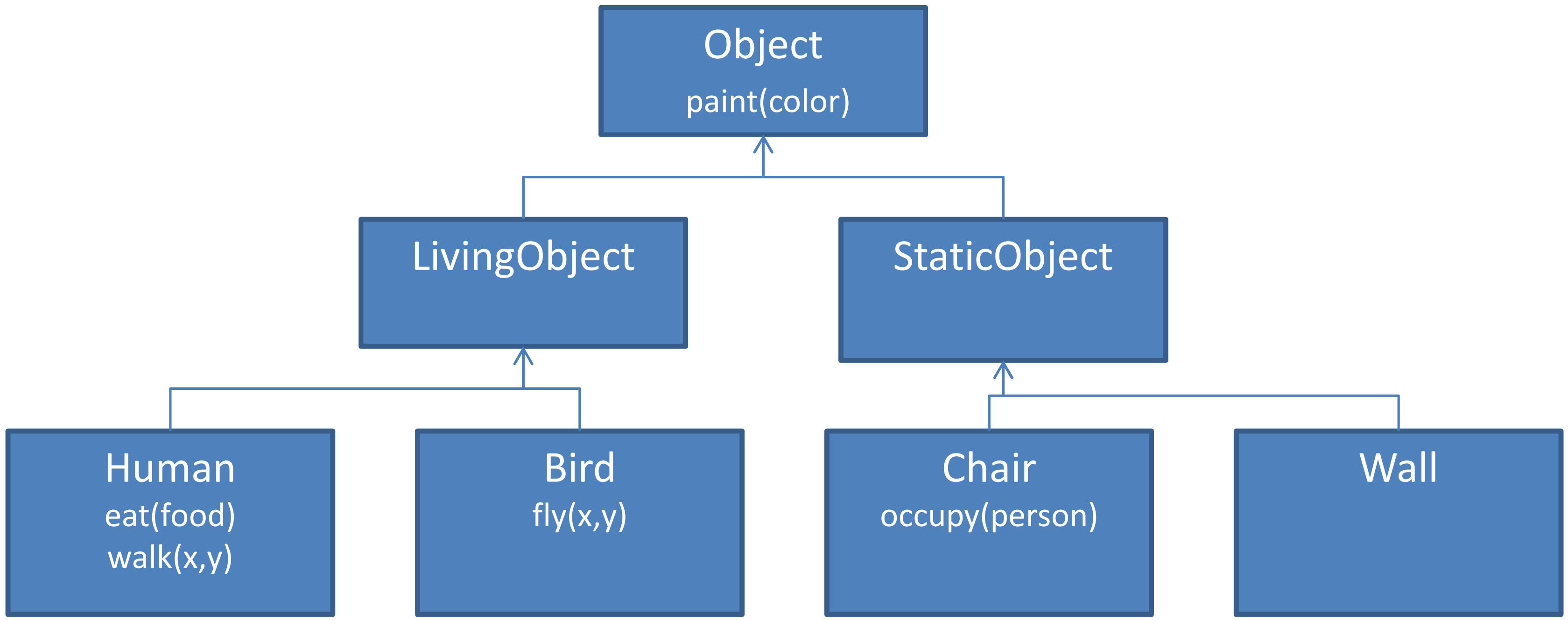
The Blob (anti)pattern

- Modelling our world with traditional OO methodology:



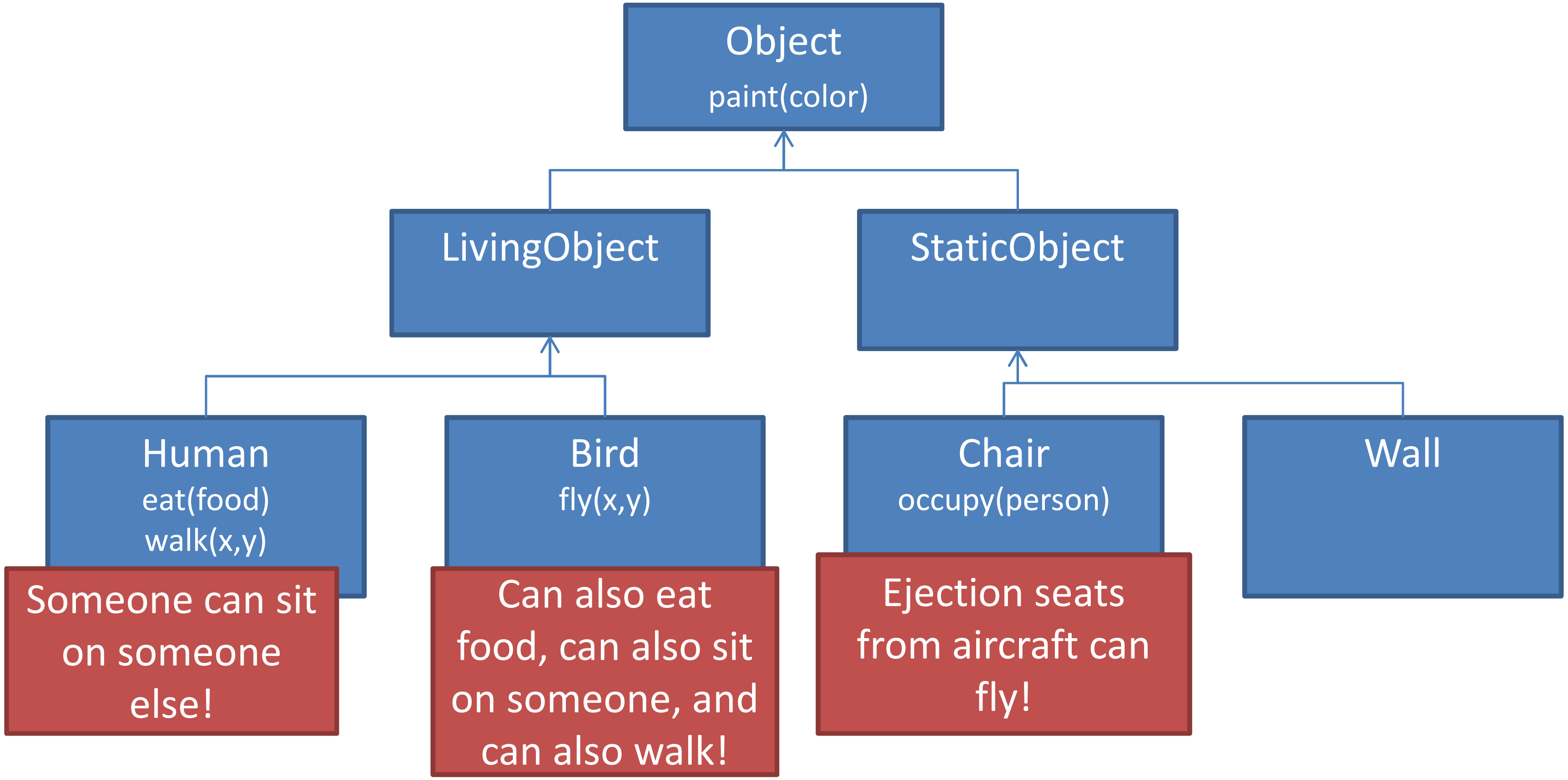
The Blob (anti)pattern

- Modelling our world with traditional OO methodology:



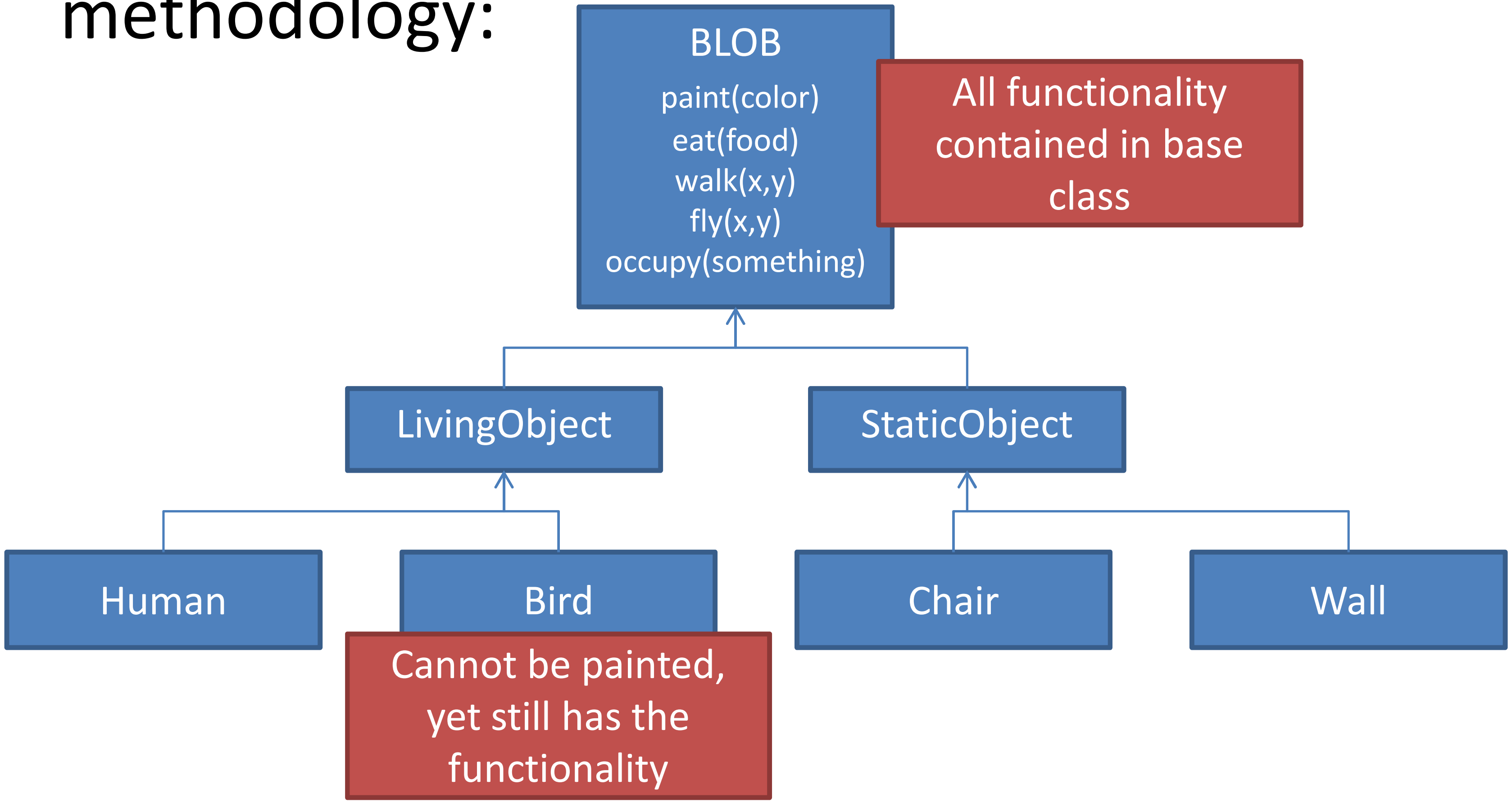
The Blob (anti)pattern

- Modelling our world with traditional OO methodology:



The Blob (anti)pattern

- Modelling our world with traditional OO methodology:



The Blob (anti)pattern

- Very common in deep class hierarchies
- Ironically, traditional OO only good for relatively shallow hierarchies

→possible solution:
component-based programming!
(also known as data-driven programming)
(also known as entity-driven programming)

Outline

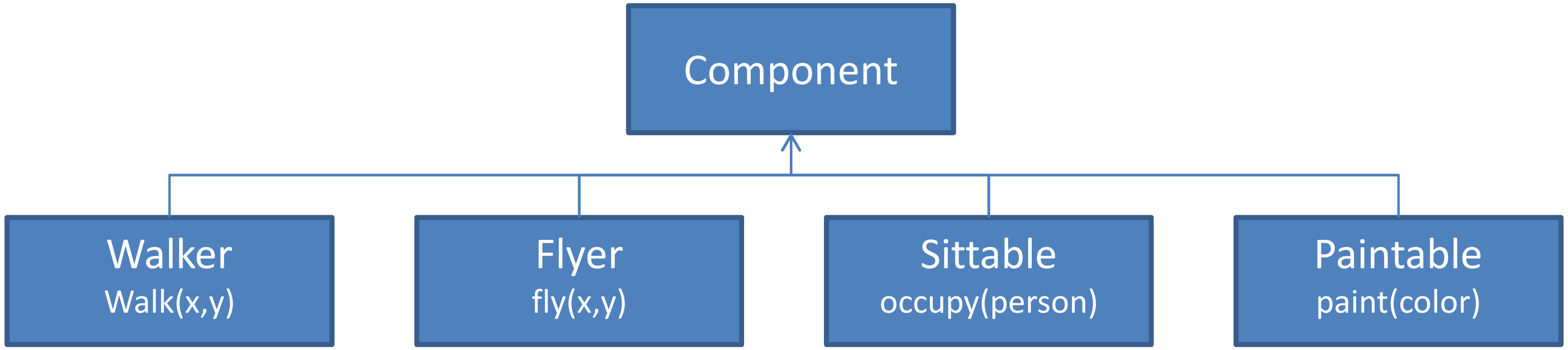
- History of programming paradigms
- Issues with object-oriented programming
- <u>Component-based programming: an alternative</u>
- Practical issues
  - Initialization
  - Communication
  - Change of mindset
- Implementation: the Cistron library
- Conclusion

Component-based programming

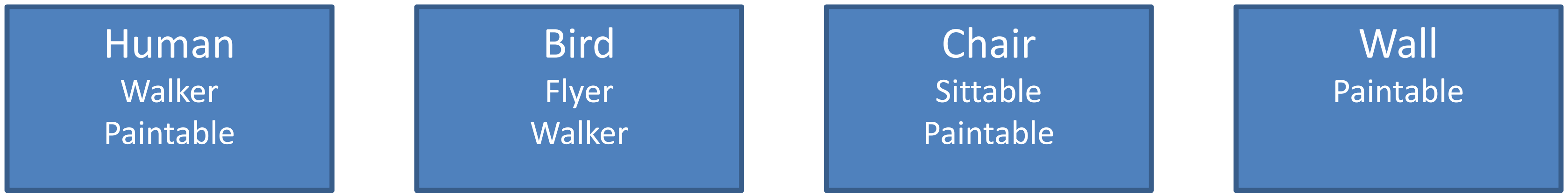
- Each object is a list of components
- Each component encapsulates a property of the object (variables and methods)
- Uses principles of OO, but avoids deep class hierarchies
- Components can communicate with each other
  - Directly (function calls)
  - Indirectly (message passing)

Our world using components

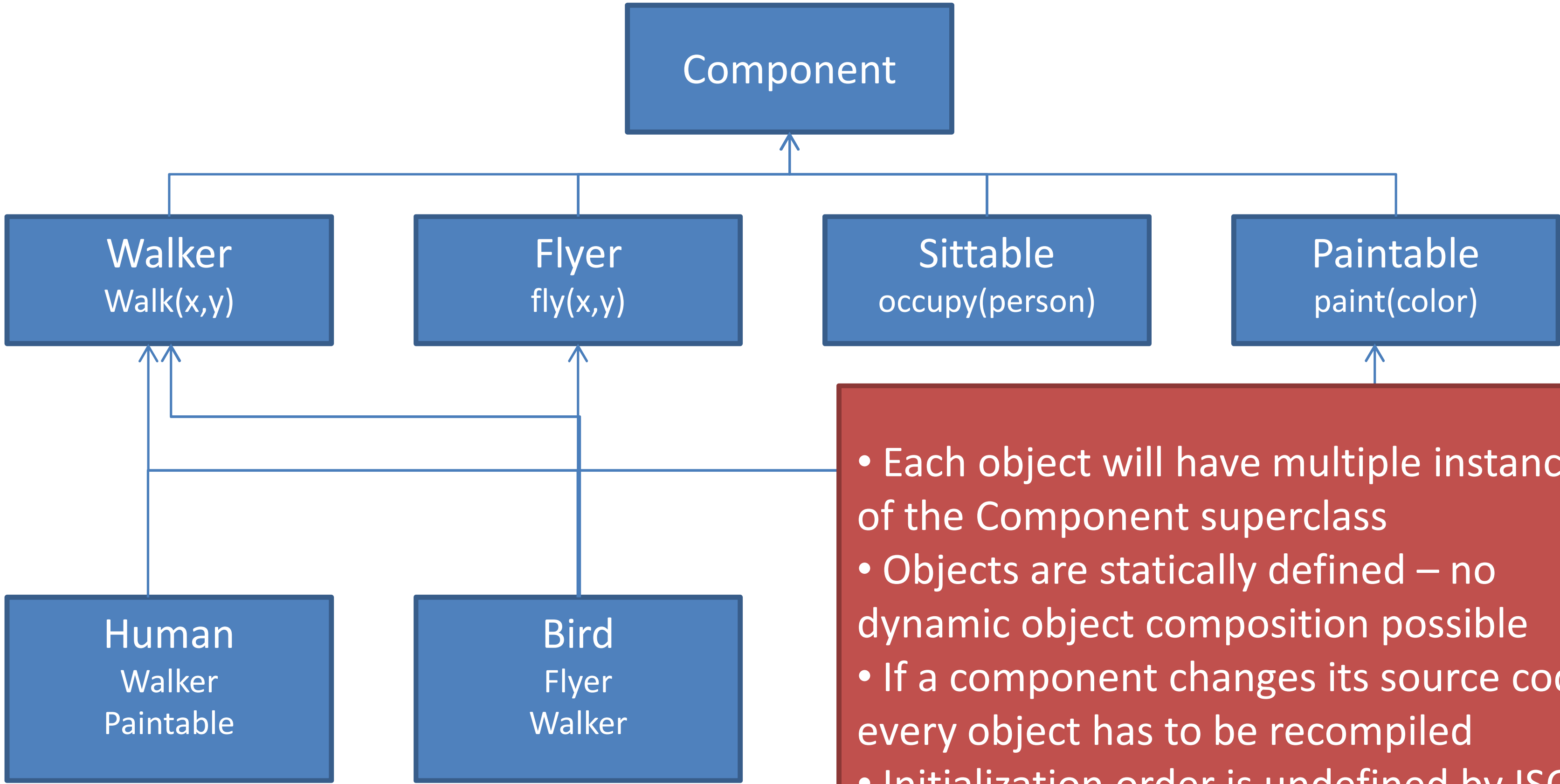
Components



Objects



Why not use multiple inheritance?



- Each object will have multiple instances of the Component superclass
- Objects are statically defined – no dynamic object composition possible
- If a component changes its source code, every object has to be recompiled
- Initialization order is undefined by ISO C++ standard
- Components cannot be added or destroyed individually

Outline

- History of programming paradigms
- Issues with object-oriented programming
- Component-based programming: an alternative
- Practical issues
  - Initialization
  - Communication
  - Change of mindset
- Implementation: the Cistron library
- Conclusion

Initialization

- Objects can be defined:
  - Statically (hard-coded in the source code)
  - Dynamically (pulled from file, e.g. xml)
- Dynamic object composition has many advantages:
  - New objects can be created without recompiling the code
  - Objects and their properties can easily be adjusted and tweaked after compilation

Initialization (2)

- XML example with dynamic loading:

```
<objects>
       <Human>
              <Walker speed=“5.0”/>
              <Paintable/>
       </Human>
       <Bird>
              <Walker speed=“2.0”/>
              <Flyer speed=“10.0”/>
       </Bird>
</objects>
```

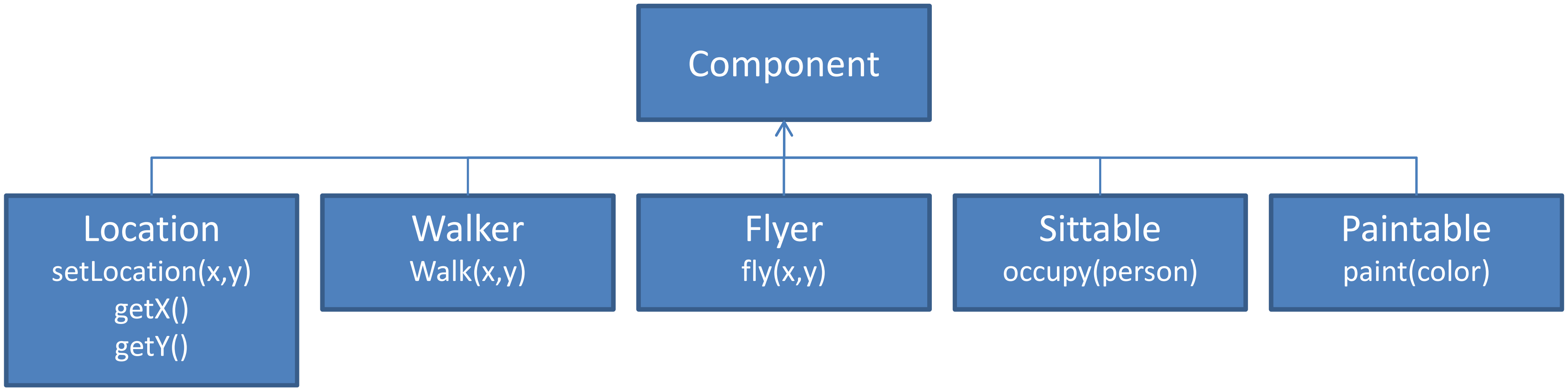
Initialization (3)

- C++ example with static loading:

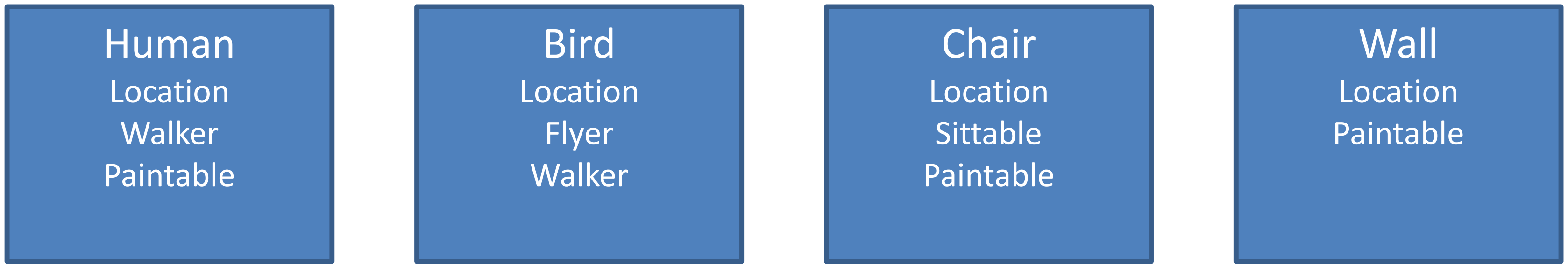
```
Object *obj = new Object();

Walker *walker = new Walker();
walker->setSpeed(5.0);
obj->addComponent(walker);

obj->addComponent(new Paintable());
```

Our world using components, revisited

Components



Objects



Communication

- Components don't function independently of each other (e.g. Walker and Flyer must be able to change the Location of the object)
- Some means of communication is necessary
- Two approaches (both viable):
  - Direct communication using dynamic cast and function calls
  - Indirect communication using message passing

Direct communication

- When added to an object, a component can request pointers to other components of a particular type in the same object
- Flyer and Walker will request Location, because they want to be able to change it
- Dynamic cast is used to cast from Component base class

Direct communication (2)

- Code example:

```
void Walker::update() {
        if (fIsWalking)
                Component *comp = requestComponent("Location");
                Location *loc = dynamic_cast<Location>(comp);
                loc->setLocation(x, y);
        }
}
```

Indirect communication

- Messages are packets of data that are sent out by one component, and received by the components that subscribed to that message type
- A message has:
  - A sender (component)
  - A set of subscribec receivers (also components)
  - An optional payload (void pointer or boost::any)
  - A range
    - Only broadcast to subscribing components in the same object
    - Broadcast to every subscribing component in the system
    - Broadcast to subscribing components in another object

Direct communication (2)

- Code example:

```
void Walker::update() {
        if (fIsWalking)
                sendMessage("Move", fNewLocation);
        }
}

void Location::receiveMessage(Message msg) {
        if (msg.type == "Move") {
                setLocation(msg.payload);
        }
}
```

Comparison

- Direct communication
  - Is fast and efficient
  - Is convenient (most closely resembles traditional OO programming practices)
  - Requires that components “know” each other – some independence is lost
- Indirect communication
  - Is highly flexible (components do not need to know of each other’s existence)
  - Requires no dynamic_cast
  - May require uglier casts if payload is necessary
  - May be impractical if a lot of complex communication is necessary
  - Is difficult to debug!

Change of mindset

- Most difficult aspect about component-based programming: **changing the way you think** about implementing stuff
- Theory is simple; changing your code-style from classical OO programming to component-based programming is **difficult**

Change of mindset (2)

- Concrete real-world objects can easily be converted to component-based entities because their properties are intuitive
- More abstract concepts, such as “Game”, “AI” and “Network” may be more difficult to adapt
- Not every part of the software needs to be a Component, but many may benefit from being one
- For example, by making your network controller part of your component system, it can:
  - Receive messages between components, and send the data across the network for synching with the other side
  - Resolve packetloss/lag issues, by broadcasting messages that tell the components about network communication conflicts

Change of mindset (3)

- When you try programming component-based for the first time, you might get stuck
  - It took me several iterations to get a quad tree right
- Don’t give up: it’s absolutely worth it!
  - In our example: adding a Swimmer component is trivial and requires no changes to Walker or Flyer at all!
- Use a good component-based framework to have most of the work (communication in particular) done for you

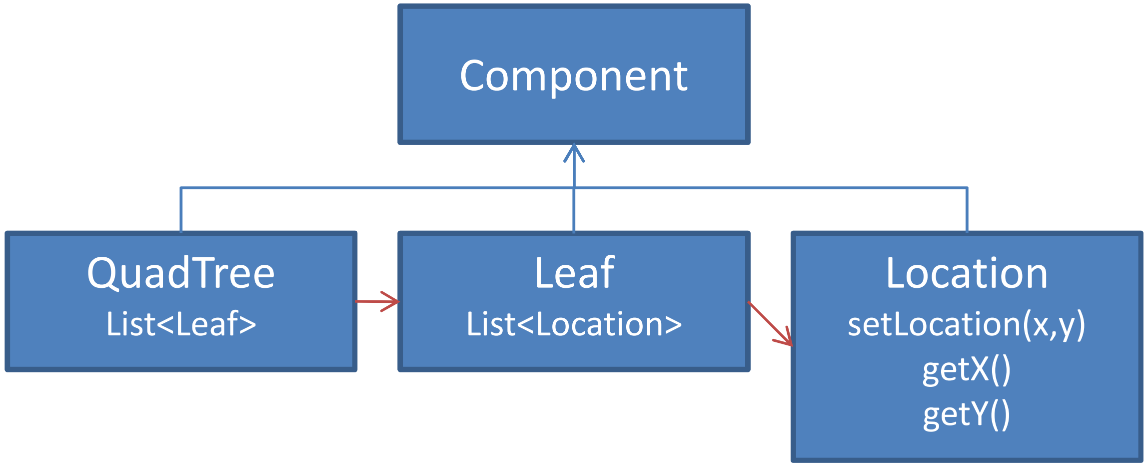
Quad tree example

- First attempt:
  - All leaves are Components
  - Object (quad tree) is a collection of Leaf components with parent links between them
  - Each Leaf component holds a list of Location components

  → Inefficient! Communication is way too slow.

Quad tree, first attempt

Components



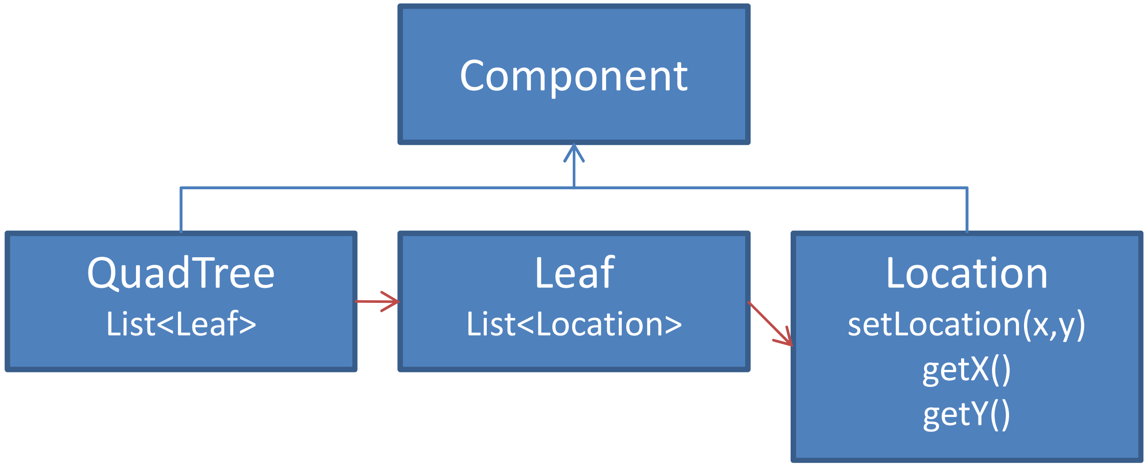
Quad tree example (2)

- Second attempt:
  - QuadTree is a component
  - QuadTree is in object which also contains NetworkController, OpenGLController, etc (global Game object)
  - Quad tree has list of Location components, and resolves collisions between them
  - Once a collision is found, both objects that contain the colliding Location components are notified through a message

    → Efficient! But what if there are many types of collisions? And many components that may influence the collision or the consequences of the collision?

Quad tree, second attempt

Components



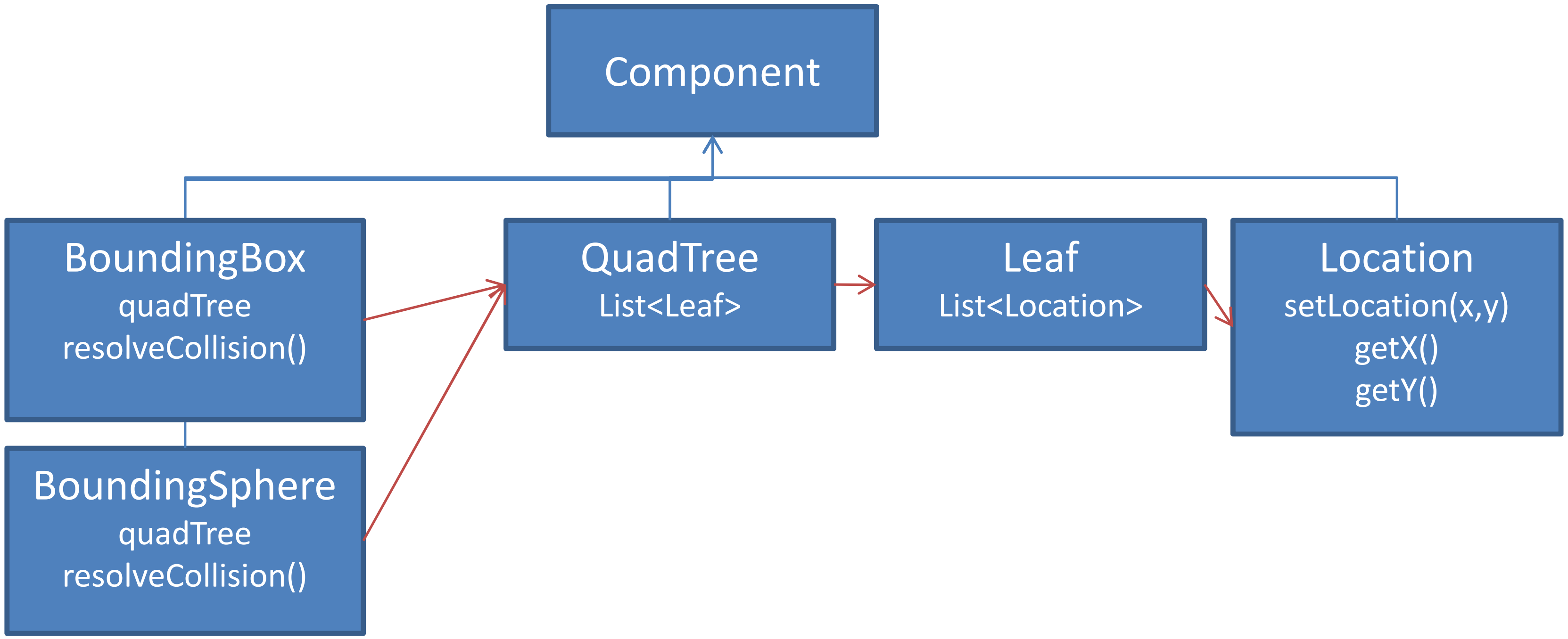
Quad tree example (3)

- Third attempt:
  - QuadTree is a component
  - QuadTree is in object which also contains NetworkController, OpenGLController, etc (global Game object)
  - BoundingBox, BoundingSphere etc are components that request both the QuadTree and the Location components
  - In the update() function, BoundingBox requests all nearby components from QuadTree, and solves everything internally

    → Efficient AND flexible!

Quad tree, third attempt

Components



Outline

- History of programming paradigms
- Issues with object-oriented programming
- Component-based programming: an alternative
- Practical issues
  - Initialization
  - Communication
  - Change of mindset
- <u>Implementation: the Cistron library</u>
- Conclusion

Cistron

- Component-based architecture aimed at game development (but not only useful for it)
- Solves the issue of communication
- Does not do dynamic initialization for you
  - You can use it statically
  - If you want to use it dynamically, use XML parser
- Open source
- Link: http://code.google.com/p/cistron

Cistron (2)

- Very light-weight and extremely efficient
  - constant time message passing!
- Platform-independent
- Depends only on Boost library
  - boost::any and boost::bind
- Uses fancy template programming to hide a lot of complexity and make the framework easy to use

Component class

- Derive from this class to make a Component
- Implements following functions:
  - void addedToObject();
    - “Constructor”, for when the component is added to the engine, and can request/send messages and components
  - void requestMessage(string msg, callbackFun);
    - Request a message. Specify a callback function to be called when such a message is received (can be member function).

Component class (2)

- Functions:
  - void requestComponent(string name, callbackFun)
    - If a component of this type is added or removed, the callback function is called to notify this component of this event.
  - void sendMessage(string msg);
    void sendMessage(MessageId msgId);
    - Send a message out through the system. The message can either be defined as:
      - a string, which will require searching through a hash table to find the subscribed receivers
      - a MessageId (long), which is an index into a vector for constant-time search for subscribed receivers

ObjectManager class

- ObjectManager is responsible for:
  - Passing messages between objects
  - Keeping track of all objects in the system, and their components
  - Notifying components that other components they requested are added/removed from the system
  - Only functions available:
    - ObjectId createObject();
    - void addComponent(ObjectId, Component*);

Conclusions

- Component-based programming can be a very viable alternative to traditional OO programming when:
  - Deep OO hierarchies are expected
  - Dynamic composition of objects is useful
- Implementation can be tricky
- … but in the end, it's often worth it!
  - Many games are designed this way nowadays
- Use **Cistron** for a quick start!

http://code.google.com/p/cistron