Software Design
Models, Tools & Processes

Lecture 4-5: Construction Phase

Cecilia Mascolo

Realization of Use Cases

- What is the link between the use cases and the classes in the class diagram(s)?

- How do we make sure that what is architected is compliant with our requirement analysis?
- *Use Case Realization shows how the classes realize the behaviour expressed in the use cases.*

Interaction diagrams

- Interaction diagrams record, in detail, how objects interact to perform a task.
- Mainly used to show how a system realises a use case (or a particular scenario in a use case).
- 2 types
  - Collaboration diagrams
  - Sequence diagrams
- Show almost identical information (i.e. one can often be generated from the other), so choice depends on aspect of the interaction needed to focus on

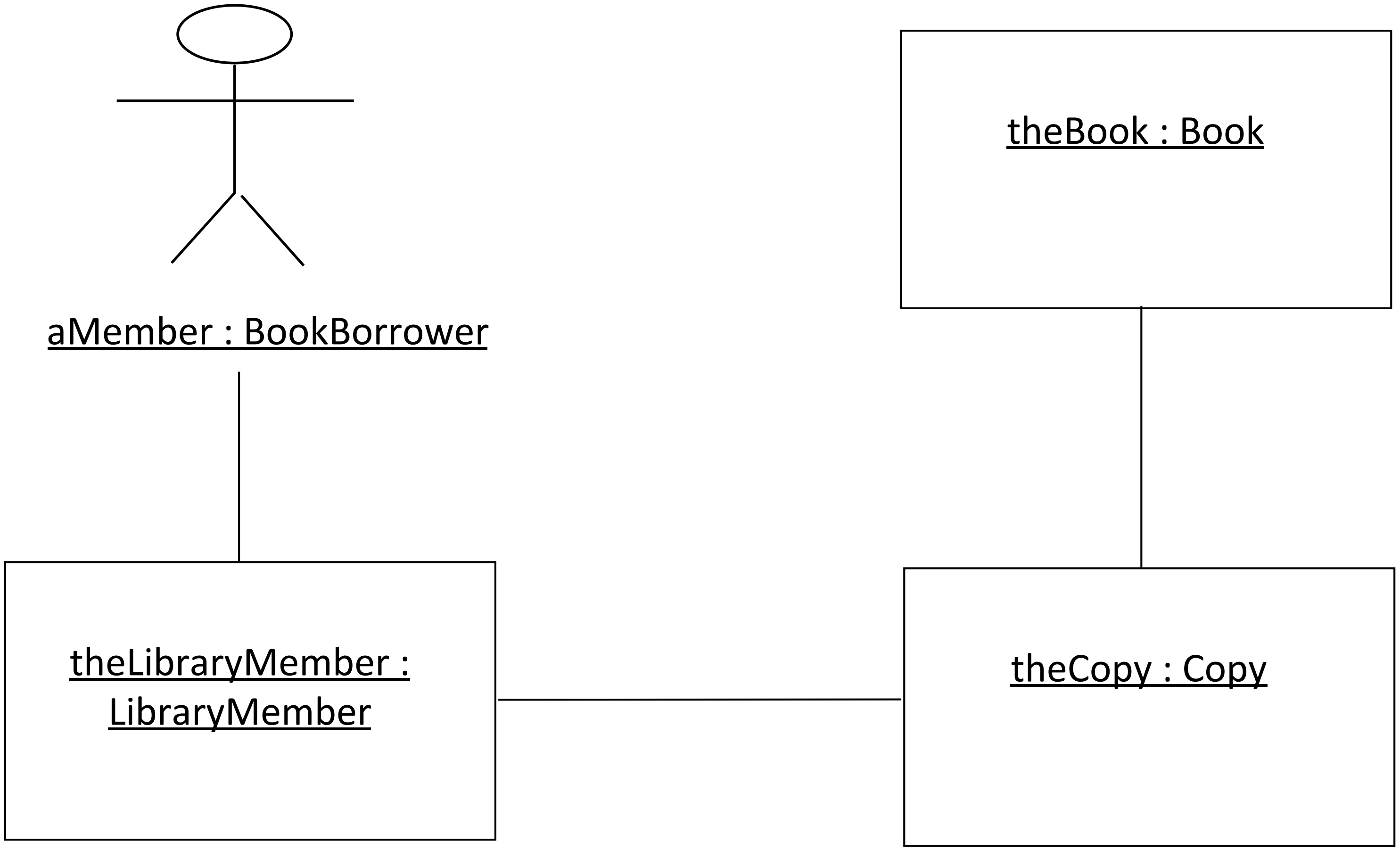
Communication diagrams

- **Communication** - the term given to the collection of objects that interact to perform some task, together with the links between them

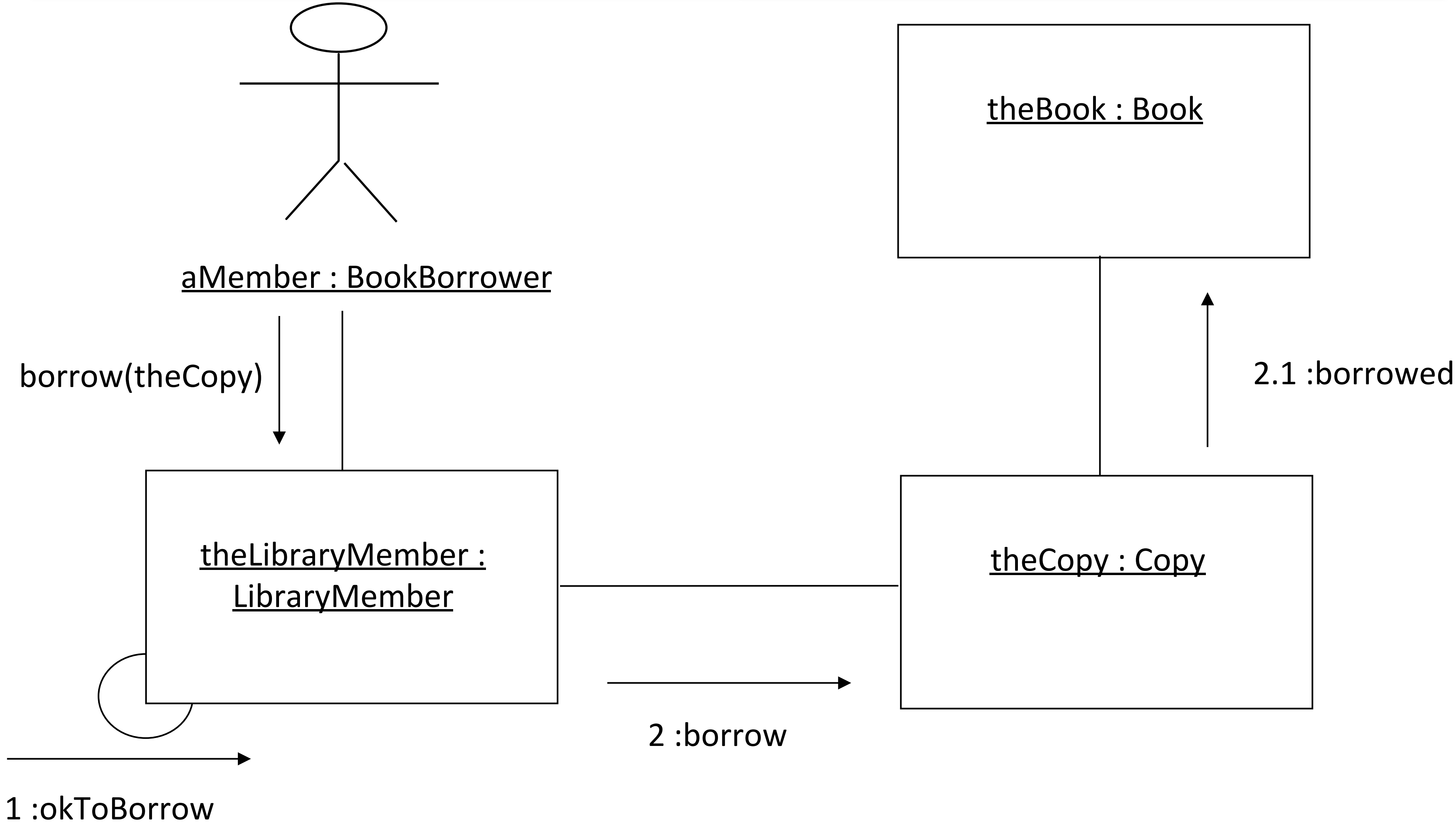
- Communication diagrams capture dynamic behaviour (*message-oriented*)

- Elements of basic communication diagrams
  - Objects
  - Links
  - Actors
  - Messages

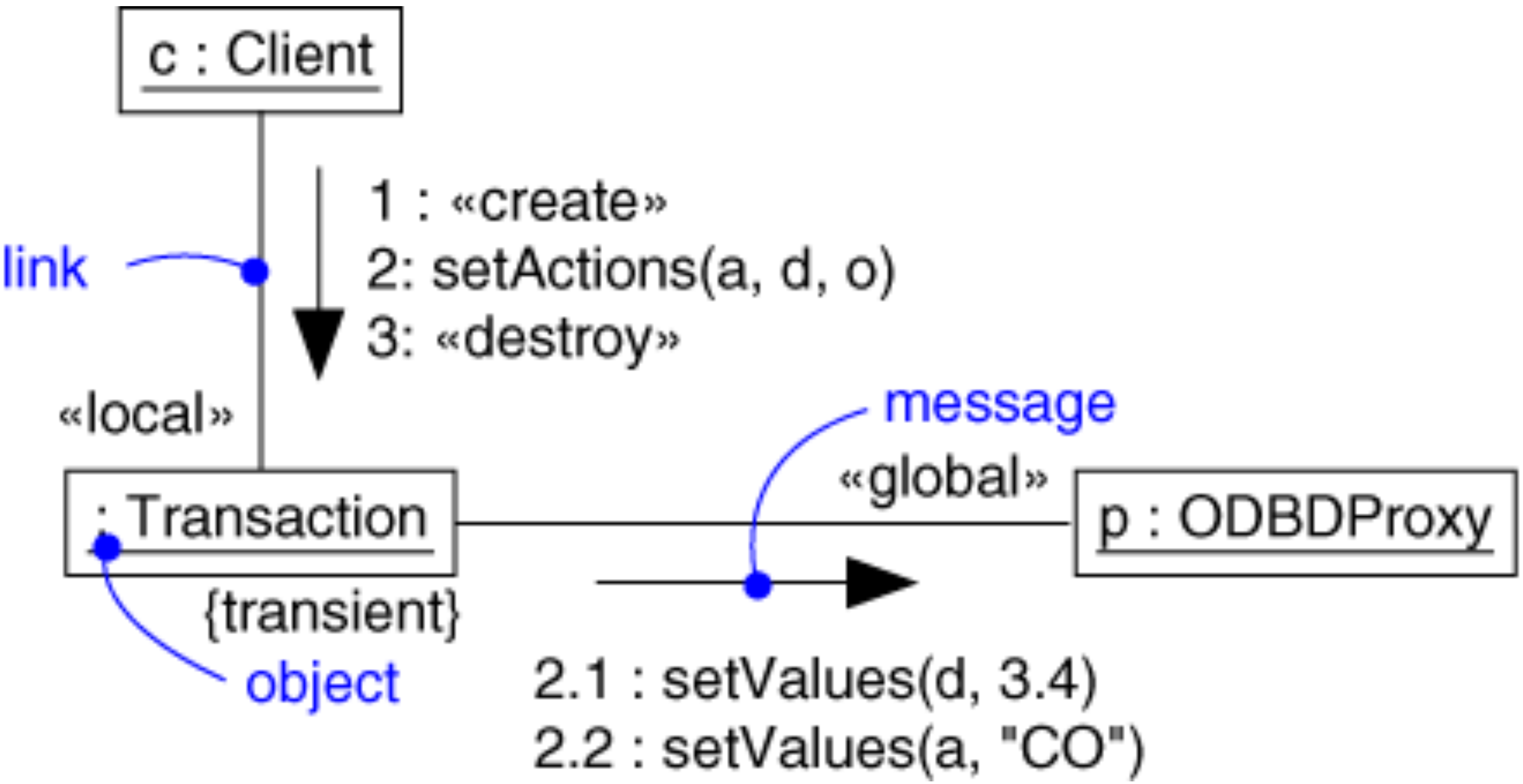
Communication diagram with no interaction



Collaboration diagram with interaction



Example communication diagram



Taken from [Booch 1999]

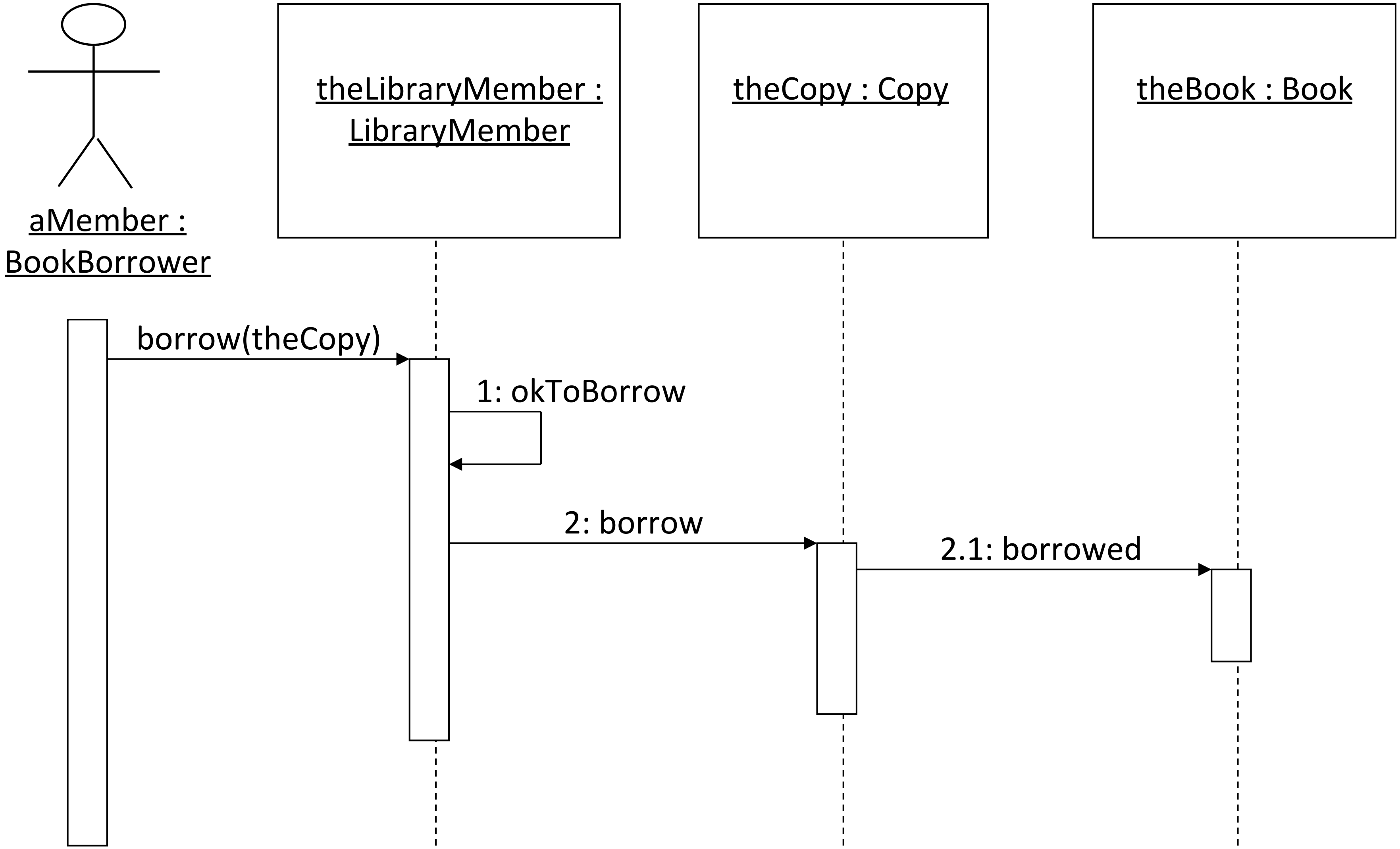
Exercise

- The use case diagram is the start of a dynamic model for the library system; the class diagram is the start of a static model for the library system - the next step is to show how the static model realises the use cases in the dynamic model
- Create communication diagrams to illustrate how classes in the model support functionality specified in the use cases
- Start by selecting a simple use case &, using UML syntax, create a communication diagram to realise it

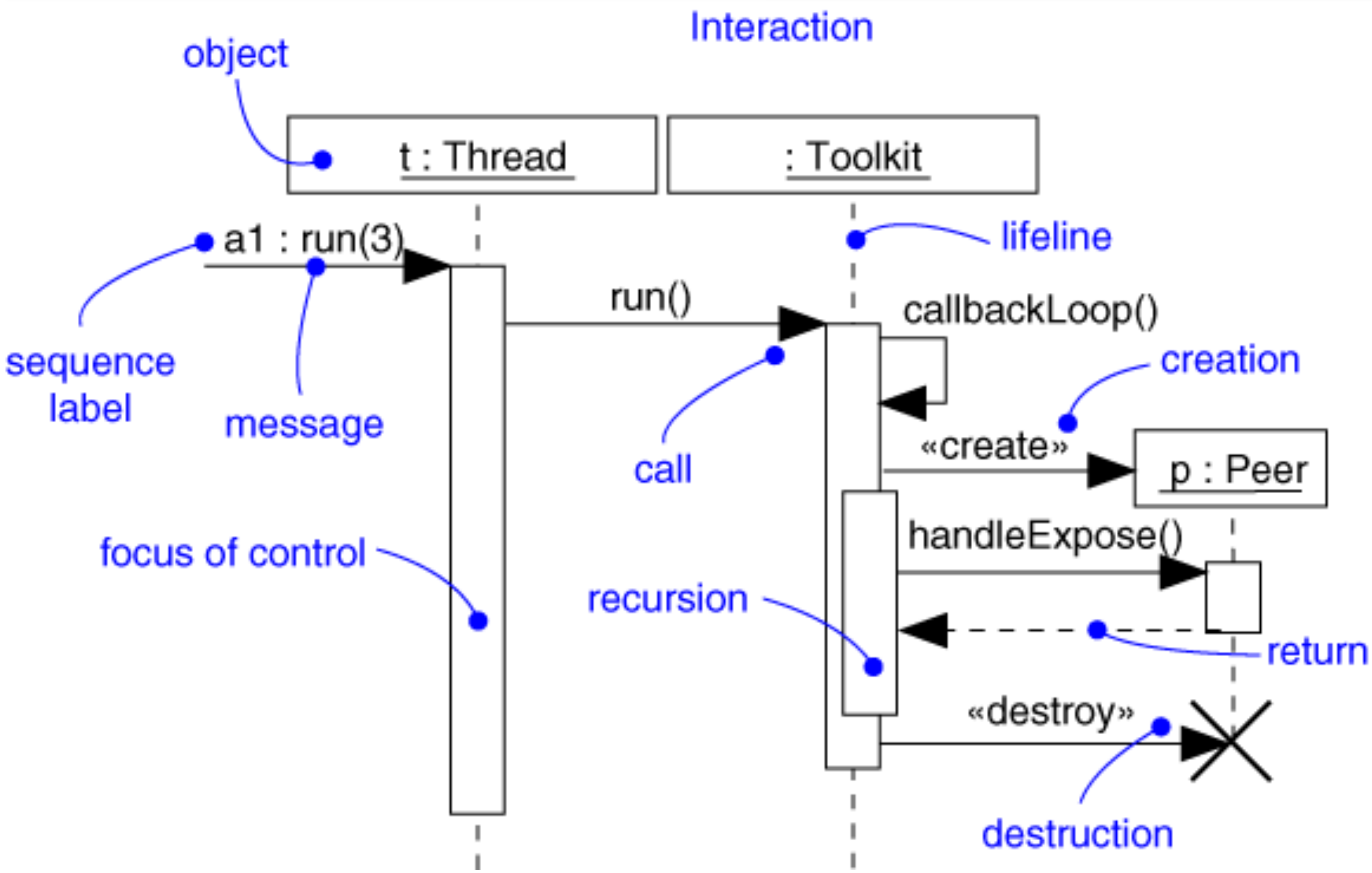
Sequence diagrams

- Sequence diagrams show object interactions arranged in a time sequence
- Sequence diagrams therefore capture dynamic behaviour (*time-oriented*)
- Elements of basic sequence diagrams
  - Objects
  - Links
  - Actors
  - Messages
  - **Object life-line**
  - **Focus of control**

Sequence diagram with interaction



Example sequence diagram



Taken from [Booch 1999]

RATIONAL SOFTWARE

Exercise

- Take the collaboration diagram you have created for part of your library system
- Turn this diagram into a sequence diagram

*** **THEN:**

- In turn, take *each* use case for your library system & create *either* a communication diagram or sequence diagram that realises the use case (aim to practice both techniques evenly)

Some notation

Interaction type	**Symbol**	**Meaning**
Synchronous or call	→	The "normal" procedural situation. Sender relinquishes control until receiver has handled the message
Return	⇠	Return from an earlier message (optional). Unblocks a synchronous send
Flat	→	Message doesn't expect a reply - control passes from sender to receiver, so only the receiver will send the next message
Asynchronous	⇀	Message doesn't expect a reply - but sender stays active & may send further messages

Loose coupling

- Coupling: links between parts of a program.
- If two classes depend closely on details of each other, they are *tightly coupled*.
- We aim for *loose coupling*.
  - keep parts of design clear & independent
  - may take several design iterations
- Loose coupling makes it possible to:
  - achieve reusability, modifiability
  - understand one class without reading others;
  - change one class without affecting others.
- Thus improves maintainability.

Responsibility-driven design

- Which class should I add a new method to?
  - Each class should be responsible for manipulating its own data.
  - The class that owns the data should be responsible for processing it.
- Leads to low coupling & "client-server contracts"
  - Consider every object as a server
  - Improves reliability, partitioning, graceful degradation

Interfaces as specifications

- Define method *signatures* for classes to interact
  - Include parameter and return types.
  - Strong separation of required functionality from the code that implements it (information hiding).
- Clients interact independently of the implementation.
  - But clients can choose from alternative implementations.

Causes of error situations

- Incorrect implementation.
  - Does not meet the specification.
- Inappropriate object request.
  - E.g., invalid index.
- Inconsistent or inappropriate object state.
  - E.g. arising through class extension.
- Not always programmer error
  - Errors often arise from the environment (incorrect URL entered, network interruption).
  - File processing often error-prone (missing files, lack of appropriate permissions).

Defensive programming

- Client-server interaction.
  - Should a server assume that clients are well-behaved?
  - Or should it assume that clients are potentially hostile?
- Significant differences in implementation required.
- Issues to be addressed
  - How much checking by a server on method calls?
  - How to report errors?
  - How can a client anticipate failure?
  - How should a client deal with failure?

Argument values

- Arguments represent a major ‘vulnerability’ for a server object.
  - Constructor arguments initialize state.
  - Method arguments often control behavior.
- Argument checking is one defensive measure.
- How to report illegal arguments?
  - To the user? ***Is*** there a human user?
    Can the user do anything to solve the problem?
    If not solvable, what should you suggest they do?
  - To the client object:
    return a diagnostic value, or throw an exception.

Example of diagnostic return

```
public boolean removeDetails(String key)
{
    if(keyInUse(key)) {
        ContactDetails details =
                        (ContactDetails) book.get(key);
        book.remove(details.getName());
        book.remove(details.getPhone());
        numberOfEntries--;
        return true;
    }
    else {
        return false;
    }
}
```

Diagnostic OK

Diagnostic not OK

Client response to diagnostic

- Test the return value.
  - Attempt recovery on error.
  - Avoid program failure.
- Ignore the return value.
  - Cannot be prevented.
  - Likely to lead to program failure.
- Exceptions are preferable.

Error response and recovery

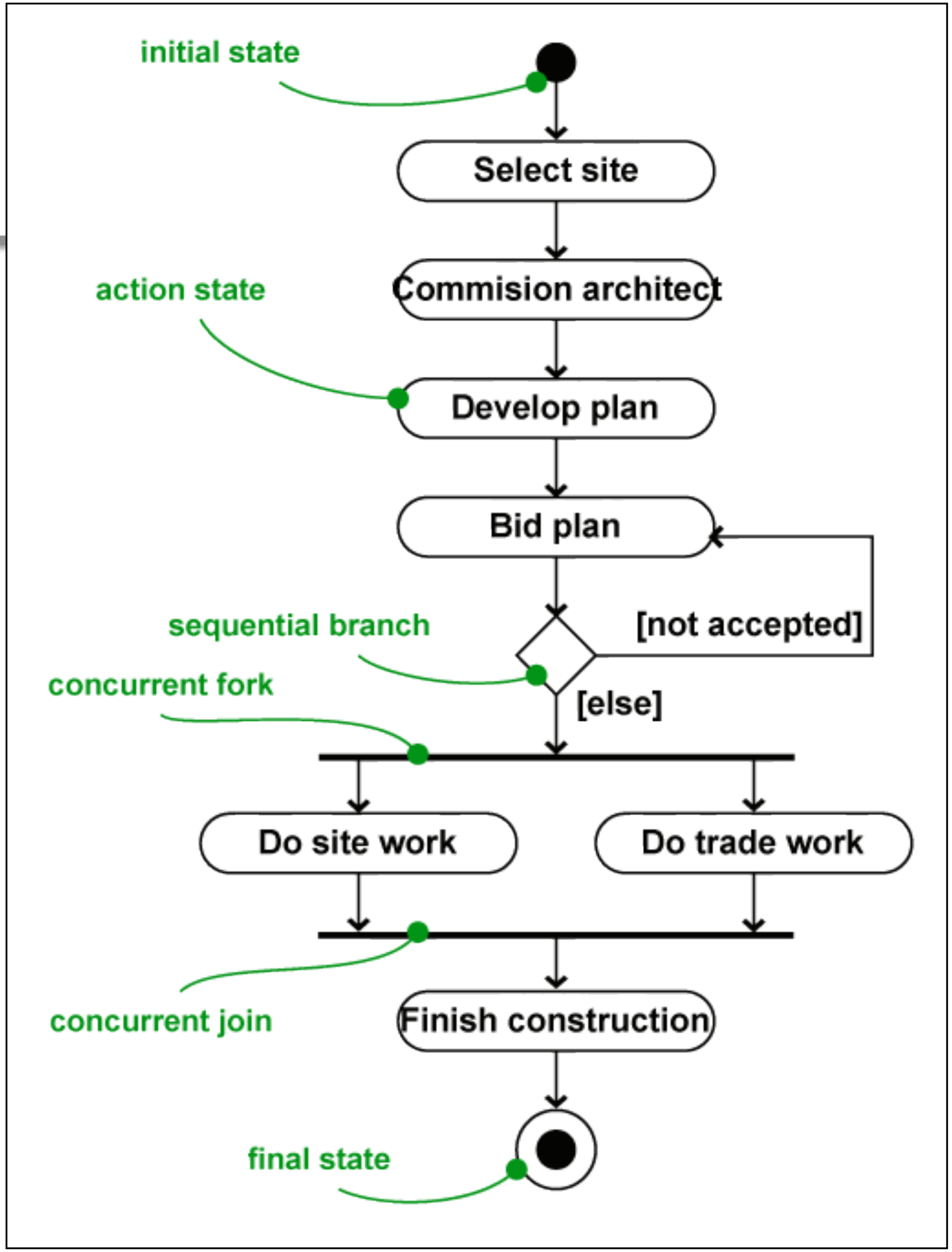
- Clients should take note of error notifications.
  - Check return values.
  - Don’t ‘ignore’ exceptions.
- Include code to attempt recovery.
  - Will often require a loop.

Error avoidance

- Clients can often use server query methods to avoid errors.
  - More robust clients mean servers can be more trusting.
  - Unchecked exceptions can be used.
  - Simplifies client logic.
- May increase client-server coupling.

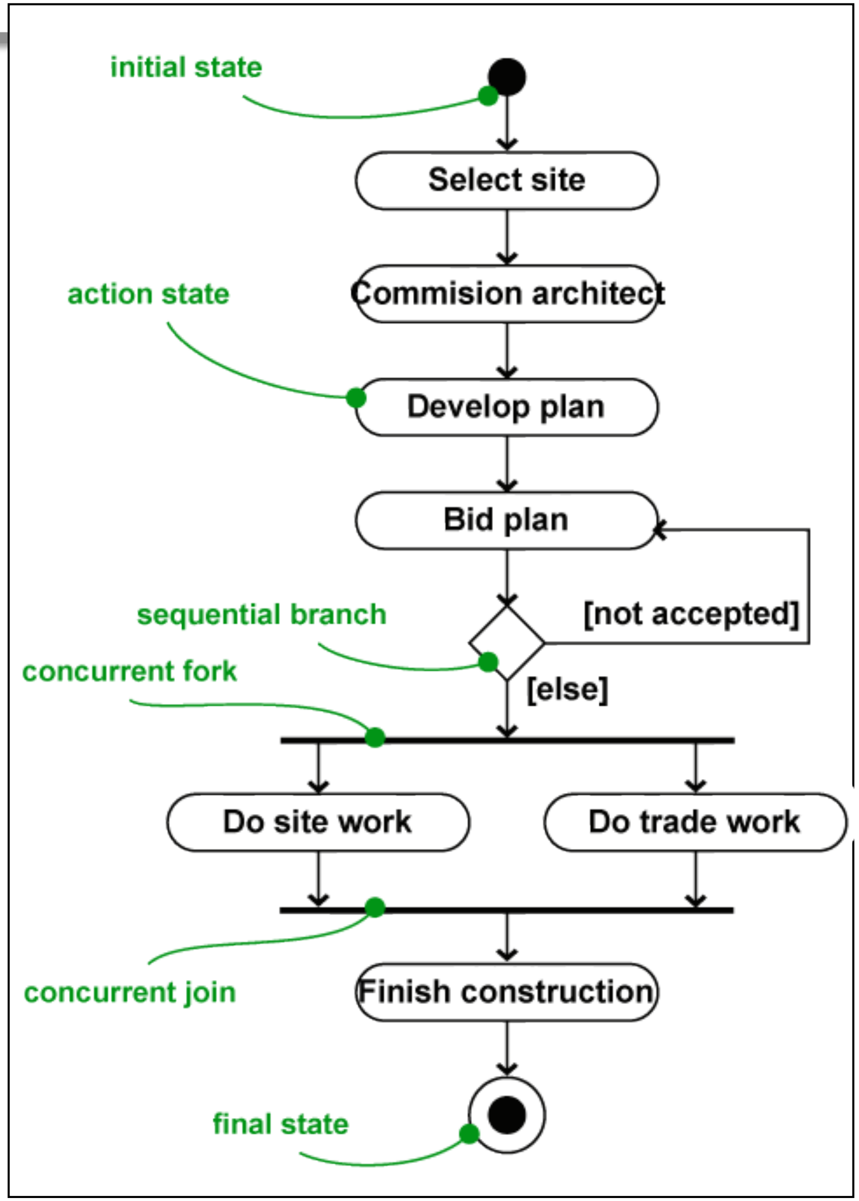
Construction inside Objects

UML Activity diagram



UML Activity diagram

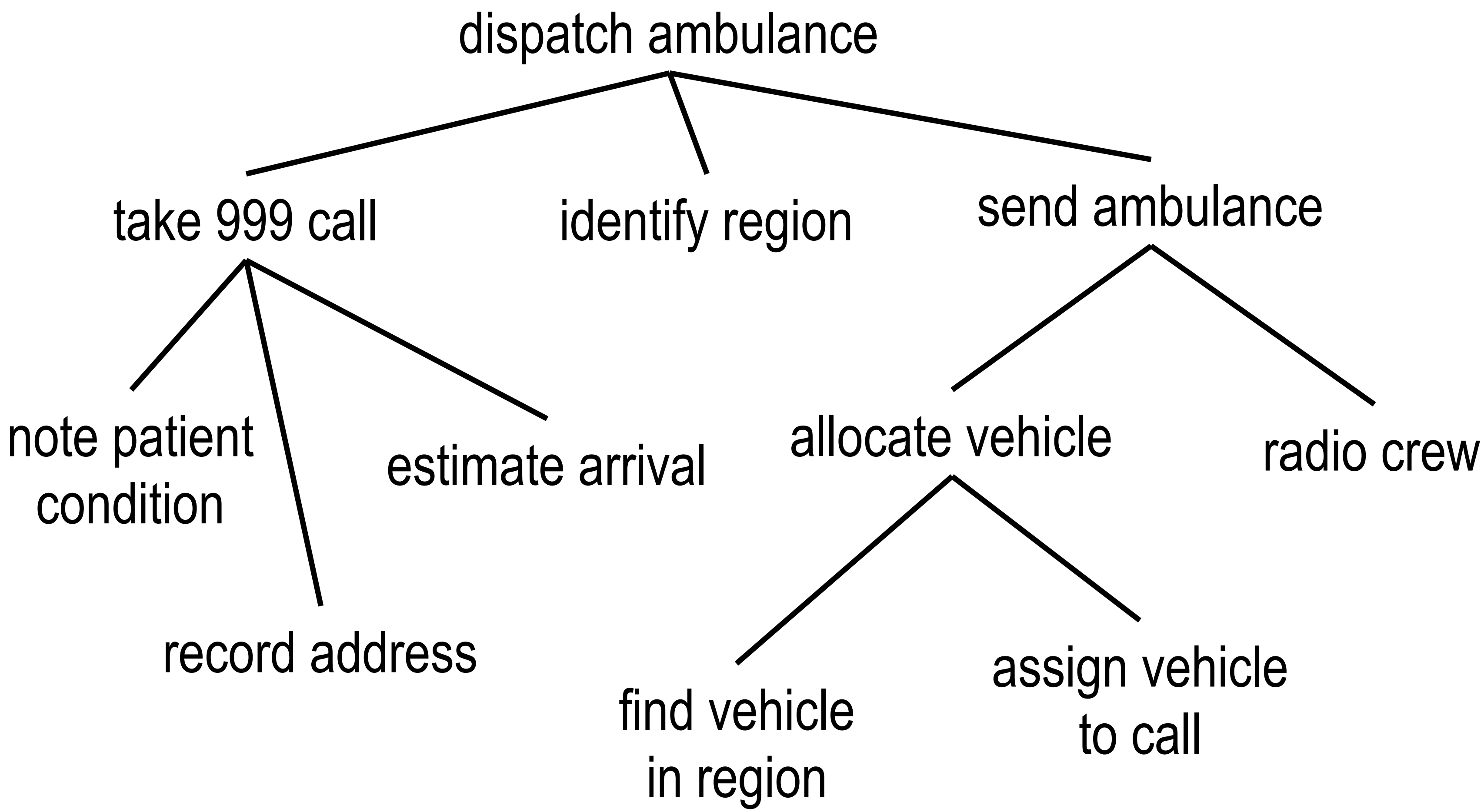
- Like flow charts
  - Activity as action states
- Flow of control
  - transitions
  - branch points
  - concurrency (fork & join)
- Illustrate flow of control
  - high level - e.g. workflow
  - low level - e.g. lines of code



Pioneers – Edsger Dijkstra

- Structured Programming
  - 1968, Eindhoven
- Why are programmers so bad at understanding dynamic processes and concurrency?
  - (ALGOL then – but still hard in Java today!)
- Observed that “go to” made things worse
  - Hard to describe what state a process has reached, when you don’t know which process is being executed.
- Define process as nested set of execution blocks, with fixed entry and exit points

Top-down design & stepwise refinement



Bottom-up construction

- Why?
  - Start with what you understand
  - Build complex structures from well-understood parts
  - Deal with concrete cases in order to understand abstractions
- Study of expert programmers shows that real software design work combines top-down and bottom up.

Modularity at code level

- Is this piece of code (class, method, function, procedure … “routine” in McConnell) needed?
- Define what it will do
  - What information will it hide?
  - Inputs
  - Outputs (including side effects)
  - How will it handle errors?
- Give it a good name
- How will you test it?
- Think about efficiency and algorithms
- Write as comments, then fill in actual code

Modularity in non-OO languages

- Separate source files in C
  - Inputs, outputs, types and interface functions defined by declarations in "header files".
  - Private variables and implementation details defined in the "source file"
- Modules in ML, Perl, Fortran, …
  - Export publicly visible interface details.
  - Keep implementation local whenever possible, in interest of information hiding, encapsulation, low coupling.

Source code as a design model

- Objectives:
  - ***Accurately*** express logical structure of the code
  - ***Consistently*** express the logical structure
  - Improve ***readability***
- Good visual layout shows program ***structure***
  - Mostly based on white space and alignment
  - The compiler ignores white space
  - Alignment is the single most obvious feature to human readers.
- Like good typography in interaction design: but the "users" are other programmers!

Code as a structured model

```
public int Function_name (int parameter1, int parameter2)

// Function which doesn't do anything, beyond showing the fact
// that different parts of the function can be distinguished.

    int local_data_A;
    int local_data_B;

    // Initialisation section
    local_data_A = parameter1 + parameter2;
    local_data_B = parameter1 - parameter2;
    local_data_B++;

    // Processing
    while (local_data_A < 40) {
        if ( (local_data_B * 2) > local_data_A ) then {
            local_data_B = local_data_B – 1;
        } else {
            local_data_B = local_data_B + 1;
        }
        local_data_C = local_data_C + 1;
    }
    return local_data_C;
}
```

Expressing local control structure

```
while (local_data_C < 40) {
    form_initial_estimate(local_data_C);
    record_marker(local_data_B - 1);
    refine_estimate(local_data_A);
    local_data_C = local_data_C + 1;
} // end while
```

```
if ( (local_data_B * 2) > local_data_A ) then {
    // drop estimate
    local_data_B = local_data_B - 1;
} else {
    // raise estimate
    local_data_B = local_data_B + 1;
} // end if
```

Expressing structure within a line

- Whitespacealwayshelpshumanreaders
  - `newtotal=oldtotal+increment/missamount-1;`
  - `newtotal = oldtotal + increment / missamount - 1;`
- The compiler doesn't care – take care!
  - `x = 1  *  y+2  *  z;`
- Be conservative when nesting parentheses
  - `while ( (! error) && readInput() )`
- Continuation lines – exploit alignment

```
– if ( ( aLongVariableName && anotherLongOne ) |
       ( someOtherCondition() ) )
     {
      …
     }
```

Naming variables: Form

- Priority: full and accurate (*not* just short)
  - Abbreviate for pronunciation (remove vowels)
    - e.g. CmptrScnce (leave first and last letters)
- Parts of names reflect conventional functions
  - Role in program (e.g. "count")
  - Type of operations (e.g. "window" or "pointer")
  - Hungarian naming (not really recommended):
    - e.g. pscrMenu, ichMin
- Even individual variable names can exploit typographic structure for clarity
  - **`xPageStartPosition`**
  - **`x_page_start_position`**

Naming variables: Content

- Data names describe domain, not computer
  - Describe what, not just how
  - **`CustomerName`** better than **`PrimaryIndex`**
- Booleans should have obvious truth values
  - **`ErrorFound`** better than **`Status`**
- Indicate which variables are related
  - **`CustName`**, **`CustAddress`**, **`CustPhone`**
- Identify globals, types & constants
  - C conventions: g_wholeApplet, T_mousePos
- Even temporary variables have meaning
  - **`Index`**, not **`Foo`**

Structural ***roles*** of variables

- Classification of what variables do in a routine
  - Don’t confuse with data types (e.g. int, char, float)
- Almost all variables in simple programs do one of:
  - fixed value
  - stepper
  - most-recent holder
  - most-wanted holder
  - gatherer
  - transformation
  - one-way flag
  - follower
  - temporary
  - organizer
- Most common (70 % of variables) are fixed value, stepper or most-recent holder.

Fixed value

- Value is never changed after initialization
- Example: input radius of a circle, then print area
  - variable r is a *fixed value,* gets its value once, never changes after that.
- Useful to declare “final” in Java (see variable PI).

```
public class AreaOfCircle {

  public static void main(String[] args) {
   final float PI = 3.14F;
   float r;
   System.out.print("Enter circle radius: ");
   r = UserInputReader.readFloat();
   System.out.println(“Circle area is " + PI * r * r);
  }
}
```

Stepper

- Goes through a succession of values in some systematic way
  - E.g. counting items, moving through array index
- Example: loop where multiplier is used as a stepper.
  - outputs multiplication table, stepper goes through values from one to ten.

```
public class MultiplicationTable {

   public static void main(String[] args) {
    int multiplier;
    for (multiplier = 1; multiplier <= 10; multiplier++)
        System.out.println(multiplier + " * 3 = "
            + multiplier * 3);
   }
}
```

Most-recent holder

- Most recent member of a group, or simply latest input value
- Example: ask the user for input until valid.
  - Variable s is a most-recent holder since it holds the latest input value.

```
public class AreaOfSquare {

   public static void main(String[] args) {
    float s = 0f;
    while (s <= 0) {
        System.out.print("Enter side of square: ");
        s = UserInputReader.readFloat();
    }
    System.out.println("Area of square is " + s * s);
   }
}
```

Most-wanted holder

- The "best" (biggest, smallest, closest) of values seen.
- Example: find smallest of ten integers.
  - Variable smallest is a *most-wanted holder* since it is given the most recent value if it is smaller than the smallest one so far.
  - (i is a *stepper* and number is a *most-recent holder*.)

```
public class SearchSmallest {
   public static void main(String[] args) {
    int i, smallest, number;
    System.out.print("Enter the 1. number: ");
    smallest = UserInputReader.readInt();
    for (i = 2; i <= 10; i++) {
        System.out.print("Enter the " + i + ". number: ");
        number = UserInputReader.readInt();
        if (number < smallest) smallest = number;
    }
    System.out.println("The smallest was " + smallest);
   }
}
```

Verifying variables by role

- Many student program errors result from using the same variable in more than one role.
  - Identify role of each variable during design
- There are opportunities to check correct operation according to constraints on role
  - Check stepper within range
  - Check most-wanted meets selection criterion
  - De-allocate temporary value
  - Use compiler to guarantee final fixed value
- Either do runtime safety checks (noting efficiency tradeoff), or use language features.

Type-checking as modeling tool

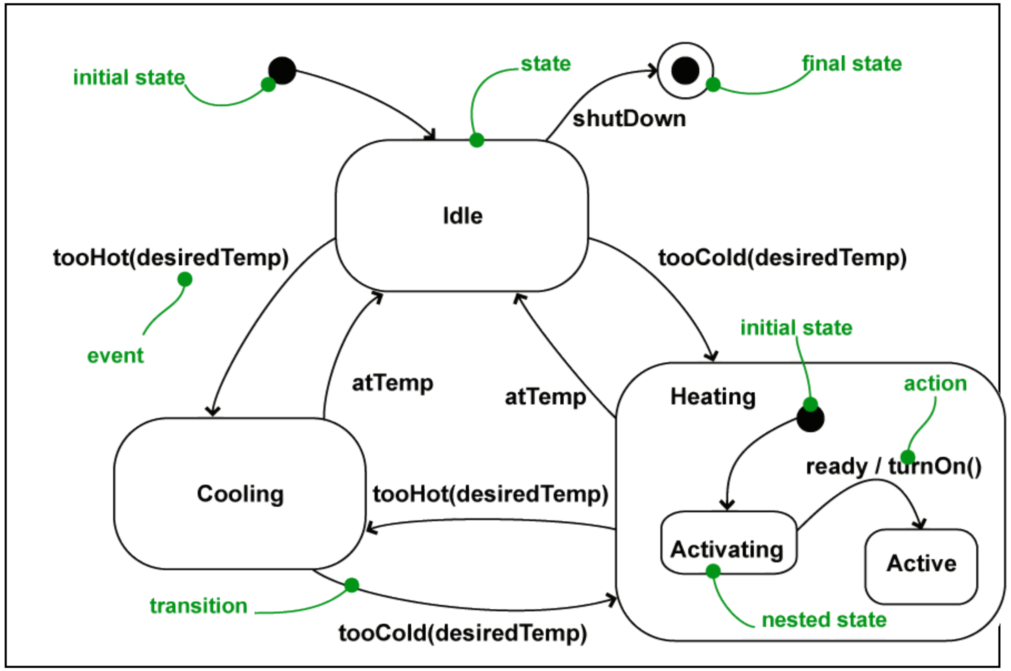
- Refine types to reflect meaning, not just to satisfy the compiler (C++ example below)
- Valid (to compiler), but incorrect, code:
  - `float totalHeight, myHeight, yourHeight;`
  - `float totalWeight, myWeight, yourWeight;`
  - `totalHeight = myHeight + yourHeight + myWeight;`
- Type-safe version:
  - `type t_height, t_weight: float;`
  - `t_height totalHeight, myHeight, yourHeight;`
  - `t_weight totalWeight, myWeight, yourWeight;`
  - `totalHeight = myHeight + yourHeight + myWeight;`

Compile error!

Construction of Data Lifecycles

UML State machine diagram

- Object lifecycle
  - data as state machine
- Harel statecharts
  - nested states
  - concurrent substates
- Explicit initial/final
  - valuable in C++
- Note inversion of activity diagram

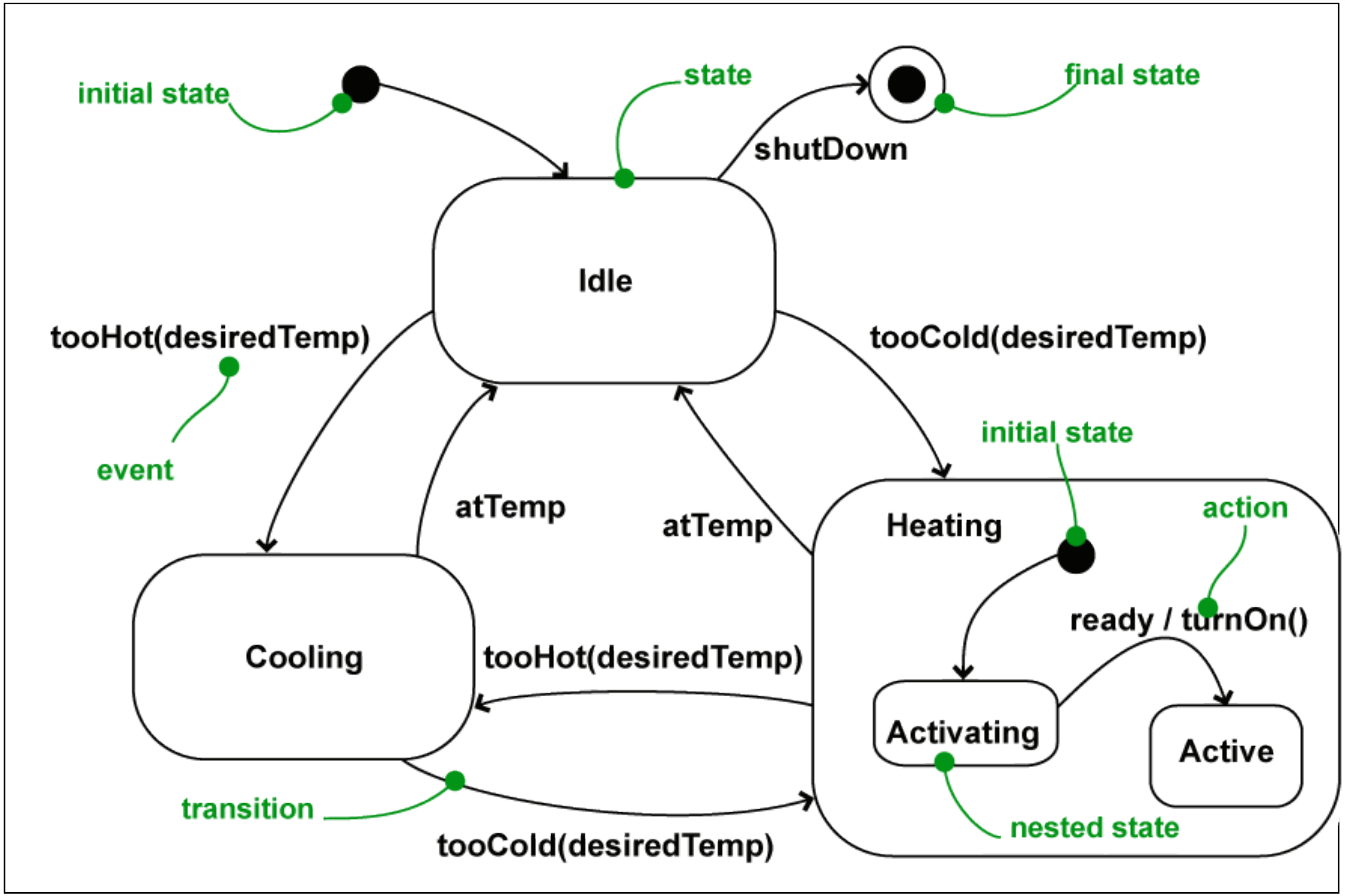


What are state machine diagrams?

- Also known as *statecharts* or *state diagrams*
- Show how an object's reaction to a message depends on its *state*
- Enables us to model an object's *decision* about what to do when it receives a message
- Used to record dependencies between the state of an object & its reaction to messages - objects of the *same class* may therefore receive the *same message*, but *respond differently*

Mostly used to model the dynamic behaviour of classes

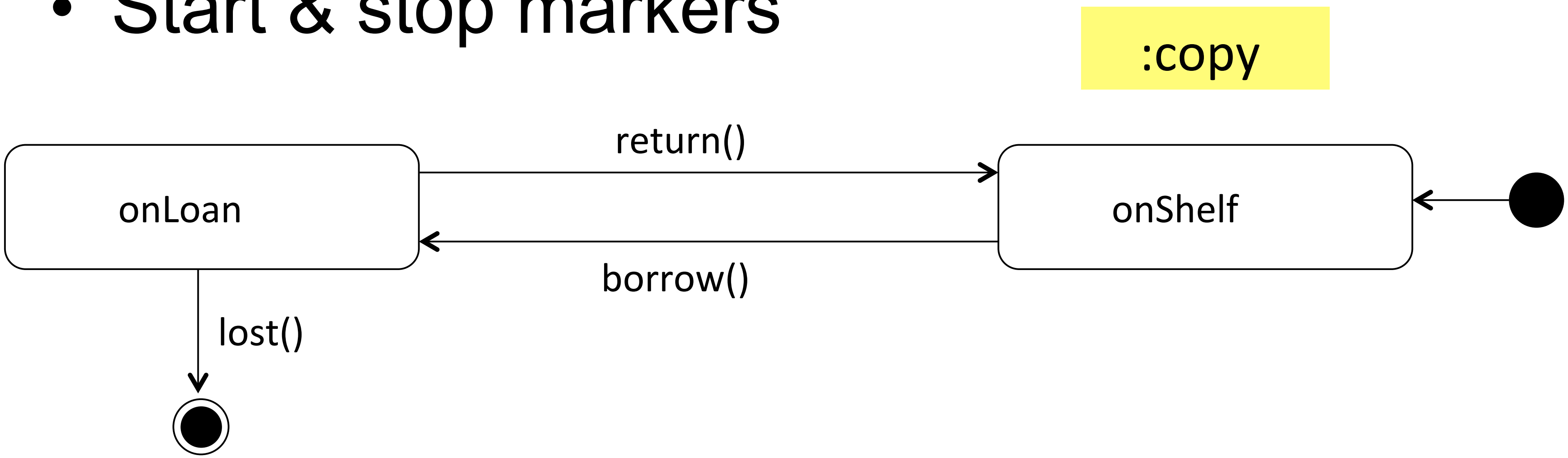
UML State Machine diagram



Elements of state machine diagrams

- States
- Events
- Transitions
- Start & stop markers

A class has **only 1** *state machine*



Thinking about states & transitions

- **State transition matrix** - a matrix with all possible states labelled on *rows* & all possible events labelled on *columns;* cells identify *next states;* responses or can be catalogued in a separate column

STATE \ EVENT	on hook	off hook	dial busy	dial idle	called party off hook		OUTPUT
idle		dial tone					quiet
dial tone	idle		busy	ringing			dial tone
busy	idle						busy tone
ringing					con-nected		rin-ging
connected							con-nected

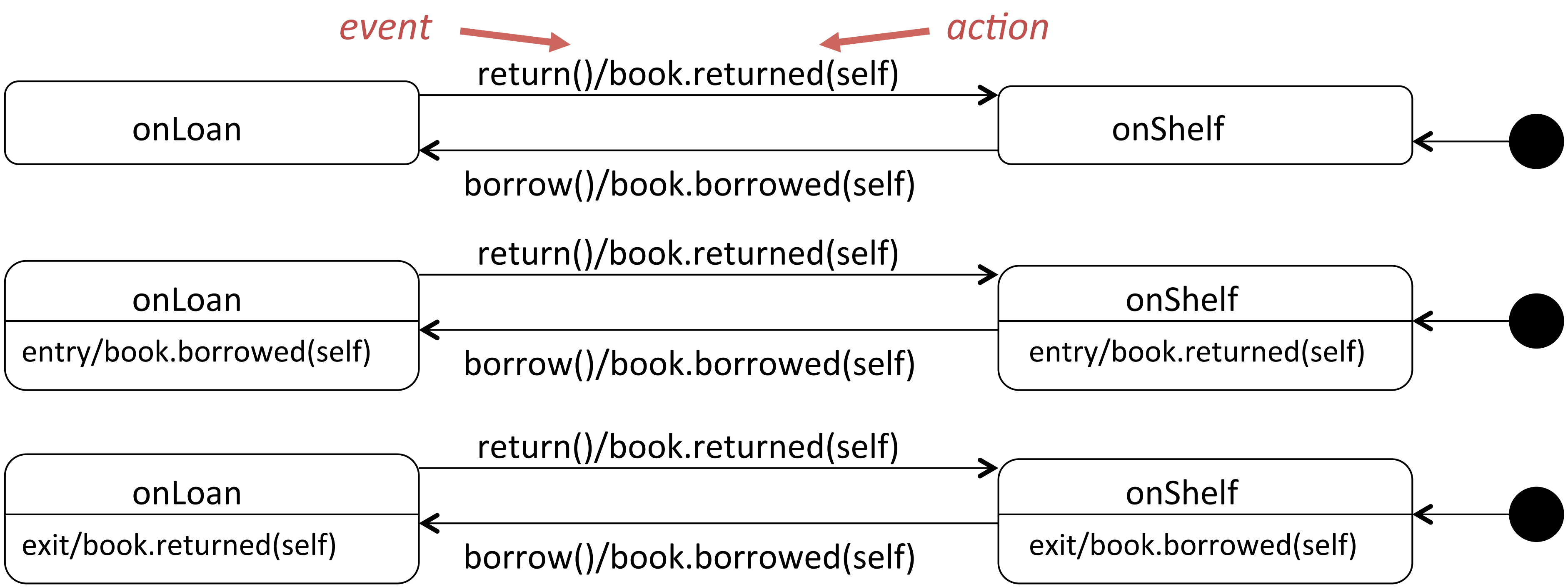
Exercise

- How many distinct states does a CD player have?

- What events occur to transition between each of these states? Remember to consider self transitions
- Sketch a simple statediagram for this CD player
- Add markers for initial & final states

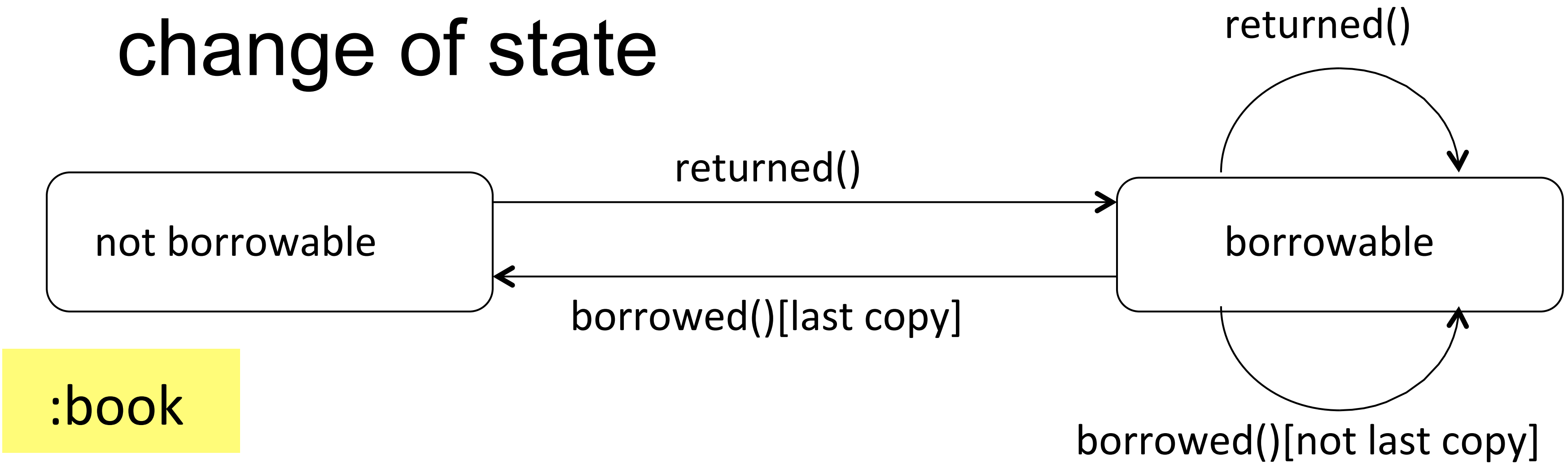
Actions & events

- *Event* - something done to an object (e.g. being sent a message) - object is the *recipient*
- *Action* - something the object does (e.g. sends a message) - object is the *instigator*



Guards

- The same event in the same state may or may not cause a change of state, depending on the object's attributes
- Conditional notation is used if the exact value of an object's attributes determines change of state



Exercise

- Take the simple state diagram for your CD player

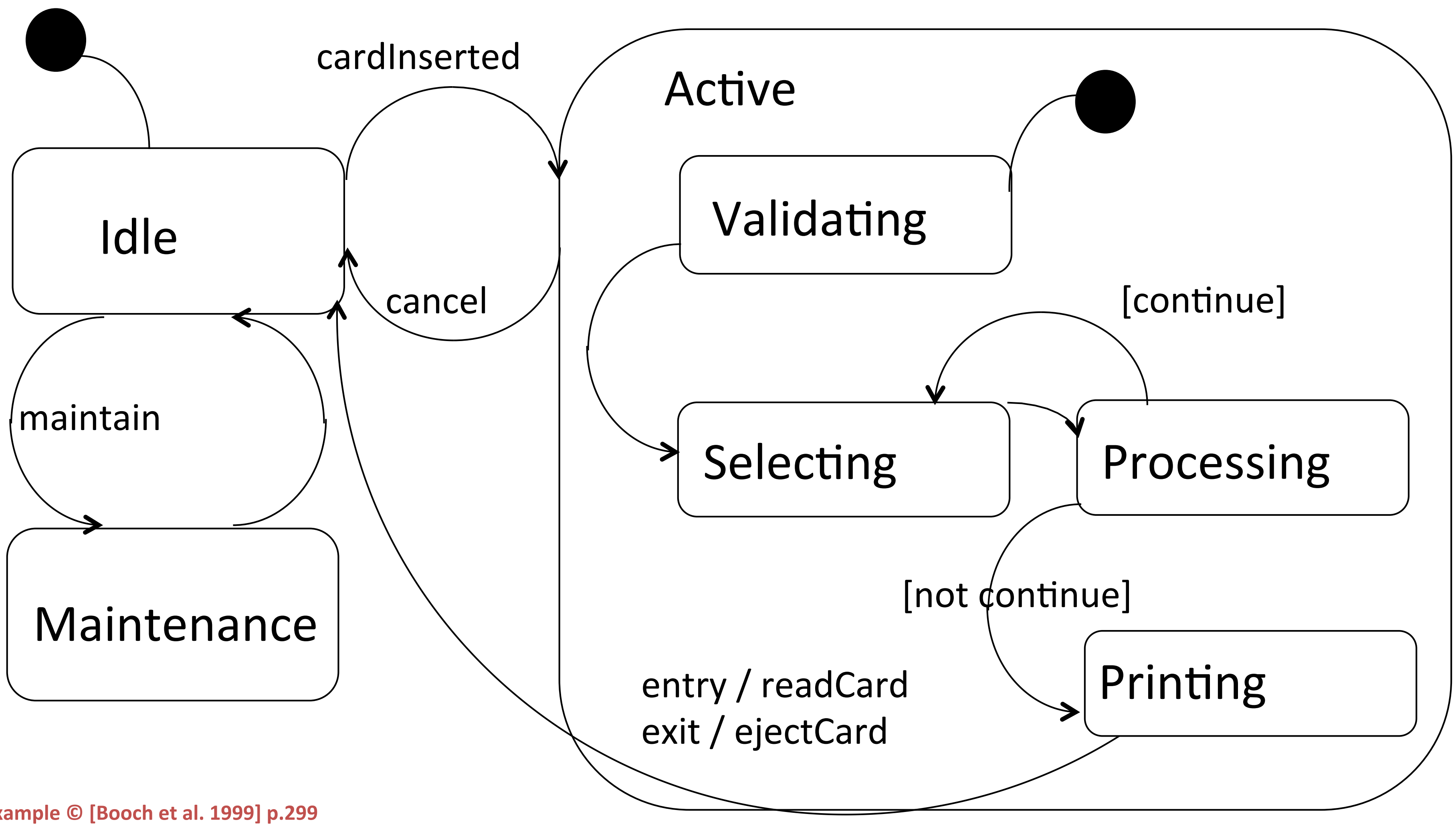
- Add useful guard conditions to some of the transitions
- List some entry & exit actions for *at least one* of the states

Substates

- States themselves can also contain internal behaviour - this can be represented as a statediagram
- Substate is a state nested inside another state
- Sequential composite/compound state - state containing a single state machine (disjoint)
- Concurrent composite/compound state - state containing 2+ state machines that execute concurrently (orthogonal)

A simple state is one that has no substructure

Sequential substates



Example © [Booch et al. 1999] p.299

Maintaining valid system state

- Pioneers (e.g. Turing) talked of proving program correctness using mathematics
- In practice, the best we can do is confirm that the state of the system is consistent
  - State of an object valid before and after operation
  - Parameters and local variables valid at start and end of routine
  - Guard values define state on entering & leaving control blocks (loops and conditionals)
  - Invariants define conditions to be maintained throughout operations, routines, loops.

Pioneers – Tony Hoare

- Assertions and proof
  - 1969, Queen’s University Belfast
- Program element behaviour can be defined
  - by a *post-condition* that will result …
  - … given a known *pre-condition*.
- If prior and next states accurately defined:
  - Individual elements can be composed
  - Program correctness is potentially provable

Formal models: Z notation

$$\begin{array}{|l}
\hline
\textit{BirthdayBook} \\
\hline
known : \mathbb{P}\, NAME \\
birthday : NAME \leftrightarrow DATE \\
\hline
known = \operatorname{dom} birthday \\
\hline
\end{array}$$

- Definitions of the *BirthdayBook* state space:
  - *known* is a set of NAMEs
  - *birthday* is a partial map from NAMEs to DATEs
- Invariants:
  - *known* must be the domain of *birthday*

Formal models: Z notation

$$
\begin{array}{|l}
\hline
\textit{AddBirthday} \\
\Delta \textit{BirthdayBook} \\
name? : NAME \\
date? : DATE \\
\hline
name? \notin known \\
birthday' = birthday \cup \{name? \mapsto date?\} \\
\hline
\end{array}
$$

- An operation to change state
  - *AddBirthday* modifies the state of *BirthdayBook*
  - Inputs are a new *name* and *date*
  - Precondition is that *name* must not be previously known
  - Result of the operation, *birthday'* is defined to be a new and enlarged domain of the *birthday* map function

Formal models: Z notation

$$
\begin{array}{|l}
\hline
\textit{Remind} \\
\Xi \textit{BirthdayBook} \\
\textit{today}? : \textit{DATE} \\
\textit{cards}! : \mathbb{P}\,\textit{NAME} \\
\hline
\textit{cards}! = \{\, n : \textit{known} \mid \textit{birthday}(n) = \textit{today}? \,\} \\
\hline
\end{array}
$$

- An operation to inspect state of *BirthdayBook*
  - This schema does not change the state of *BirthdayBook*
  - It has an output value (a set of people to send *cards* to)
  - The output set is defined to be those people whose birthday is equal to the input value *today*.

Advantages of formal models

- Requirements can be analysed at a fine level of detail.
- They are declarative (specify what the code should do, not how), so can be used to check specifications from an alternative perspective.
- As a mathematical notation, offer the promise of tools to do automated checking, or even proofs of correctness ("verification").
- They have been applied in some real development projects.

Disadvantages of formal models

- Notations that have lots of Greek letters and other weird symbols look scary to non-specialists.
  - Not a good choice for communicating with clients, users, rank-and-file programmers and testers.
- Level of detail (and thinking effort) is similar to that of code, so managers get impatient.
  - If we are working so hard,
    why aren't we just writing the code?
- Tools are available, but not hugely popular.
  - Applications so far in research / defence / safety critical
- Pragmatic compromise from UML developers
  - "Object Constraint Language" (OCL).
  - Formal specification of some aspects of the design, so that preconditions, invariants etc. can be added to models.

Language support for assertions

- Eiffel (pioneering OO language)
  - supported pre- and post-conditions on every method.
- C++ and Java support “assert” keyword
  - Programmer defines a statement that must evaluate to boolean true value at runtime.
  - If assertion evaluates false, exception is raised
- Some languages have debug-only versions, turned off when system considered correct.
  - Dubious trade-off of efficiency for safety.
- Variable roles could provide rigorous basis for fine-granularity assertions in future.

Summary

- We have illustrated how dynamics of objects can be designed through sequence and collaboration diagrams.
- We have used activity and state machine diagrams to describe object behaviour.
- We have described technique to improve code and state readability and errors avoidance.