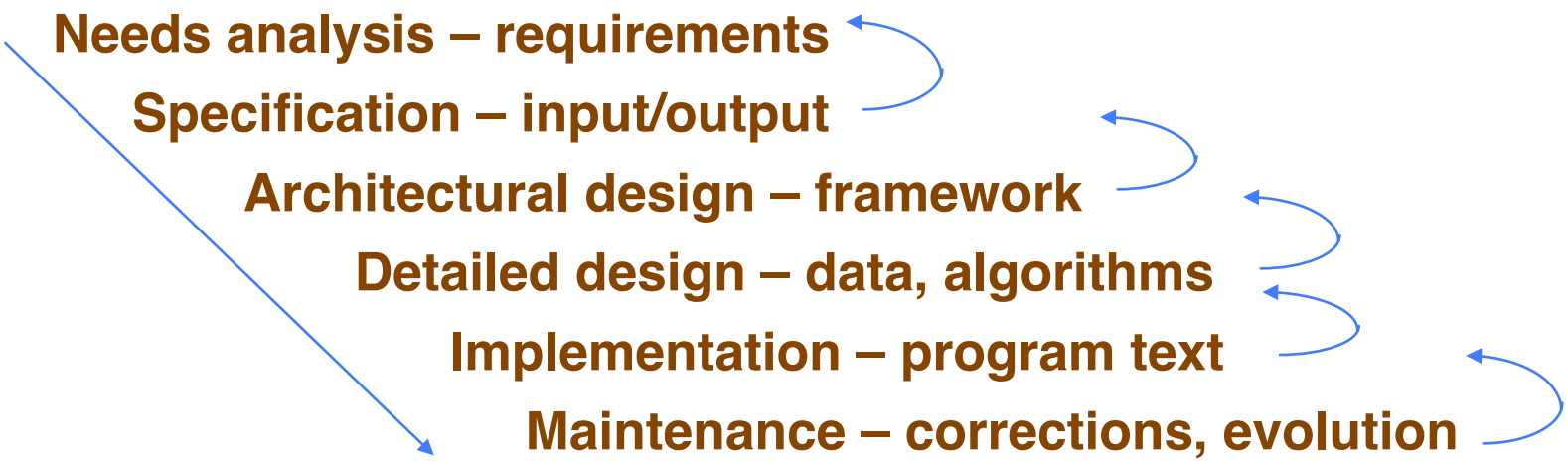
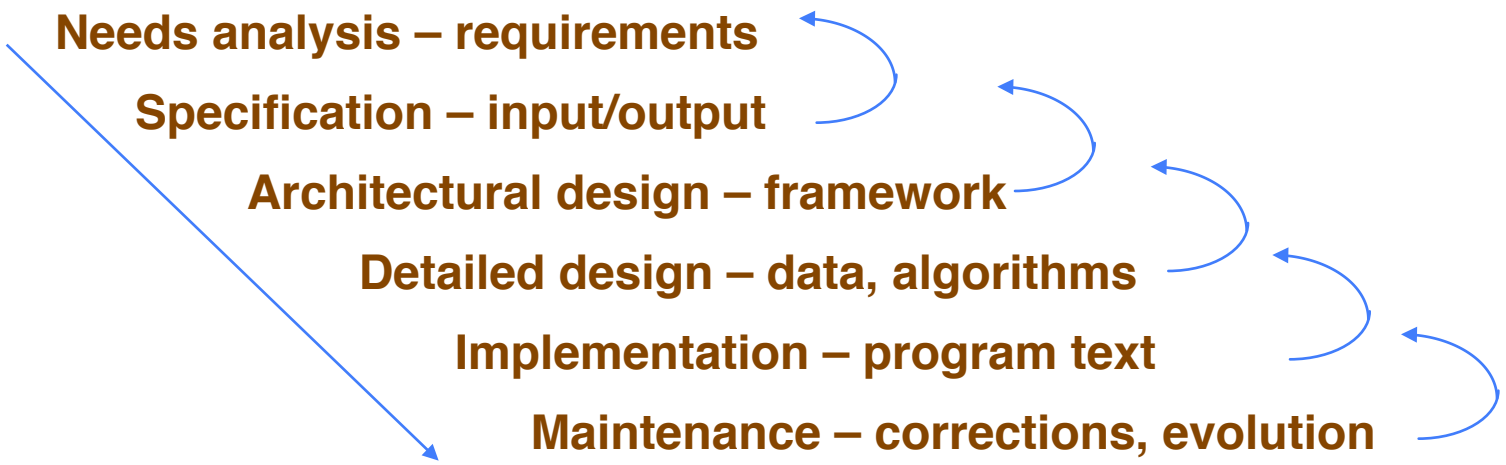
Design
Context and Principles

Waterfall Model – Software Life Cycle

Apply recursively at all levels – from system level to subprogram.
Spiral model and evolutionary development are variations

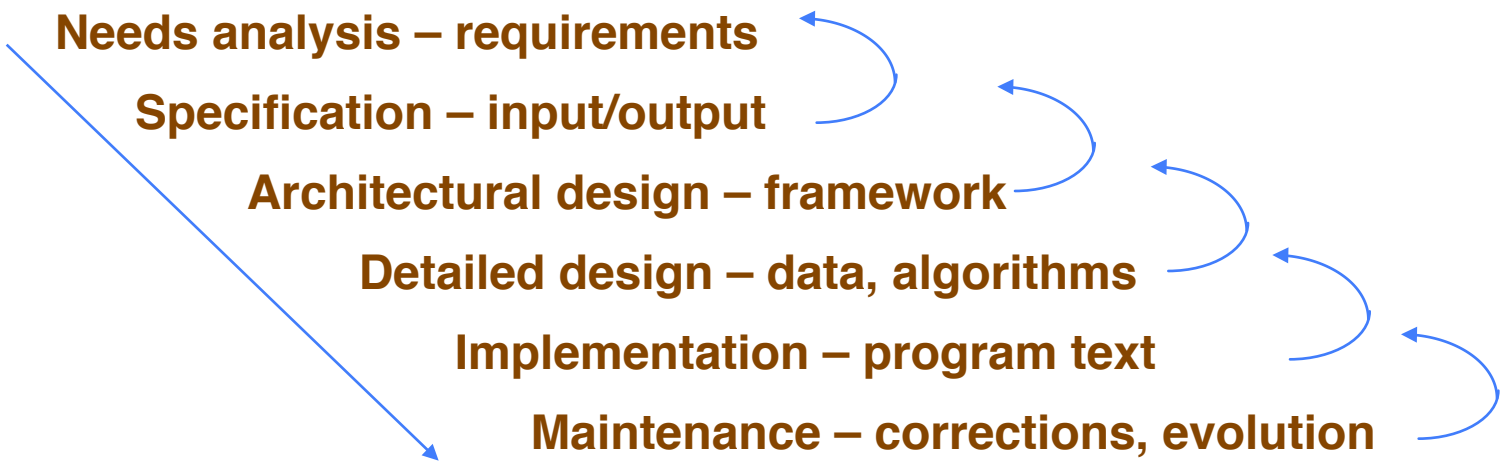


Waterfall Model – Software Life Cycle – 2



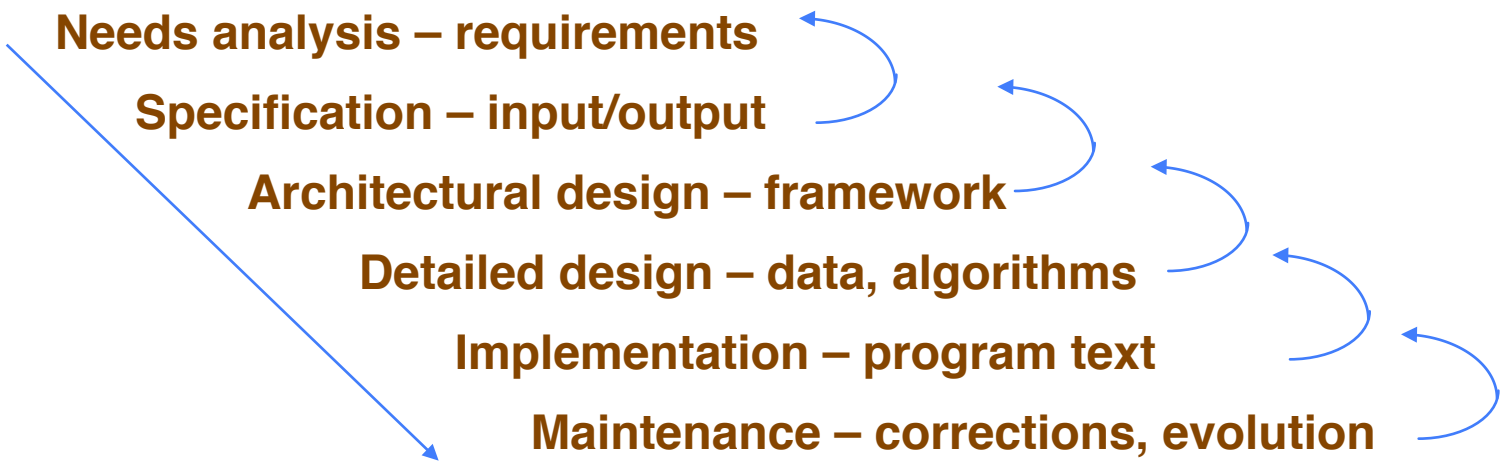
- What is produced at each level?

Waterfall Model – Software Life Cycle – 3



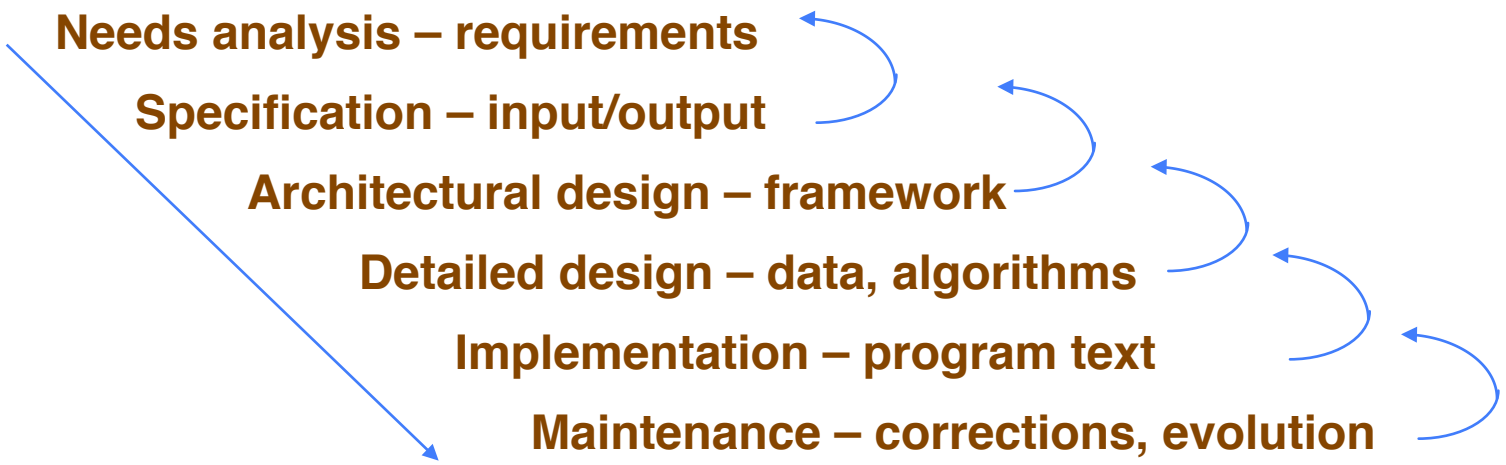
- At all stages the artifacts produced are human readable documents

Waterfall Model – Software Life Cycle – 4



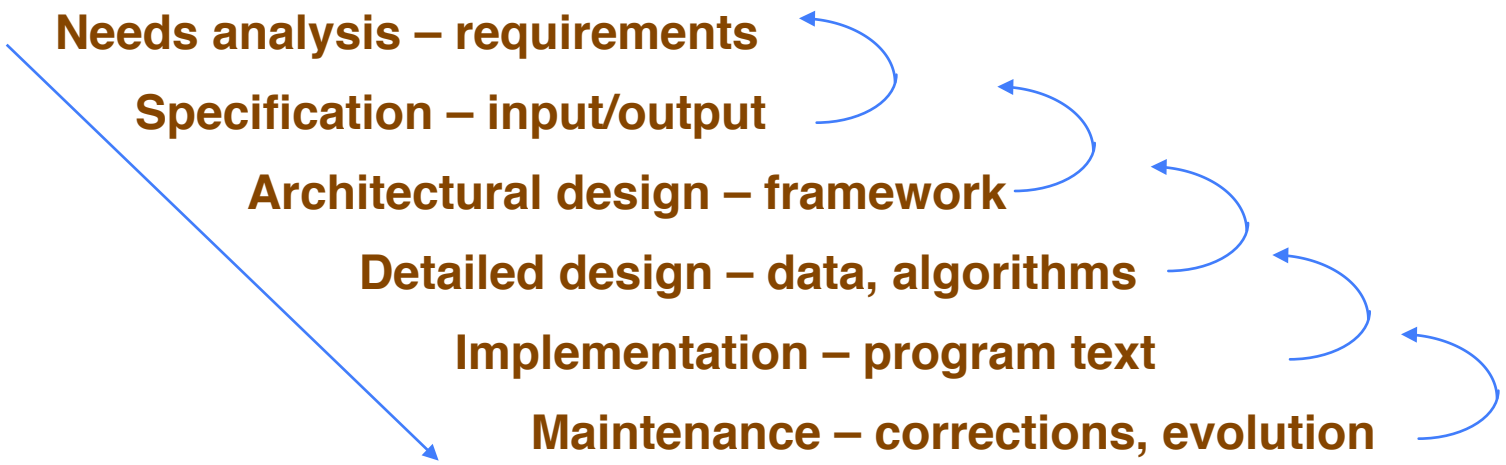
- At all stages the artifacts produced are documents
  - » **They may be formal**
    – use mathematics and programming languages

Waterfall Model – Software Life Cycle – 5



- At all stages the artifacts produced are documents
  - » **They may be formal**
    – use mathematics and programming languages
  - » **They may be informal**
    – use natural language

Waterfall Model – Software Life Cycle – 6



- At all stages the artifacts produced are documents
  - » **They may be formal**
    – use mathematics and programming languages
  - » **They may be informal**
    – use natural language
- At all times strive for correctness and precision

What is Programming?

What is Programming? – 2

- Specifying **what** to to do and **when** to do it

What is Programming? – 3

- Specifying **what** to to do and **when** to do it
- The **what** consists of the following
  - » **At the assembler level the hardwired instructions**
    - > **add, load, store, move, etc.**

What is Programming? – 4

- Specifying **what** to to do and **when** to do it
- The **what** consists of the following
  - » **At the assembler level the hardwired instructions**
    - > **Add, load, store, move, etc.**
  - » **At the Eiffel, C, Java level**
    - > **Assignment, arithmetic, read/write**
    - > **Routines from a Subprogram library, API (Application Program Interface)**

What is Programming? – 5

- The **when** consists of specifying in what order to do the "what" operations

What is Programming? – 6

- The **when** consists of specifying in what order to do the "what" operations
  - » **Control structures**

What is Programming? – 7

- The **when** consists of specifying in what order to do the "what" operations
  - » **Control structures**
    - > **What are the fundamental control structures?**

What is Programming? – 8

- The **when** consists of specifying in what order to do the "what" operations
  - » **Control structures – these are the only ones**
    - > **Sequence**
    - > **Choice**
    - > **Loop**

What is Programming? – 9

- The **when** consists of specifying in what order to do the "what" operations
  - » **Control structures – these are the only ones**
    - > **Sequence**
    - > **Choice**
    - > **Loop**
- What and when are intertwined
  - » **Changing one generally requires changing the other**

What is Design?

What is Design? – 2

- The creation of a plan.

What is Design? – 3

- The creation of a plan.
  - » **Consider design as imposing constraints on the "when" and "what" of programming.**

What is Design? – 4

- The creation of a plan.
  - » **Consider design as imposing constraints on the "when" and "what" of programming.**
  - » **From this perspective, the entire life cycle is comprised of design at various levels .**

What is Design? – 5

- The creation of a plan.
  - » **Consider design as imposing constraints on the "when" and "what" of programming.**
  - » **From this perspective, the entire life cycle is comprised of design at various levels .**
    - > **Design occurs at all the levels of the Waterfall Model from requirements to implementation**

What is Design? – 6

- Design comes from the root to designate, to name

What is Design? – 7

- Design comes from the root to designate, to name
  - » **In design one names objects and their relationships**

What is Design? – 8

- Design comes from the root to designate, to name
  - » **In design one names objects and their relationships**
  - » **The difficult part is finding the "right" objects and the "right" relationships**

What is Design? – 9

- Design comes from the root to designate, to name
  - » **In design one names objects and their relationships**
  - » **The difficult part is finding the "right" objects and the "right" relationships**
  - » **There must be a correspondence between specification and implementation**

What is Design? – 10

- Design comes from the root to designate, to name
  - » **In design one names objects and their relationships**
  - » **The difficult part is finding the "right" objects and the "right" relationships**
  - » **There must be a correspondence between specification and implementation**
    - > **The objects and relationships in the specification must correspond to the objects and relationships in the implementation**

What is Design? – 11

- Design comes from the root to designate, to name
  - » **In design one names objects and their relationships**
  - » **The difficult part is finding the "right" objects and the "right" relationships**
  - » **There must be a correspondence between specification and implementation**
    - > **The objects and relationships in the specification must correspond to the objects and relationships in the implementation**
    - > **The Direct Mapping Rule (slides on Modularity)**

What is Design? – 12

- Design comes from the root to designate, to name
  - » **In design one names objects and their relationships**
  - » **The difficult part is finding the "right" objects and the "right" relationships**
  - » **There must be a correspondence between specification and implementation**
    - > **The objects and relationships in the specification must correspond to the objects and relationships in the implementation**
    - > **The Direct Mapping Rule (slides on Modularity)**
    - > **Purpose of documentation is to show that correspondence**

Design within the Lifecycle

- Consider the constraints imposed in the software lifecycle

Design within the Lifecycle – 2

- Consider the constraints imposed in the software lifecycle
  - » **Putting together a requirements document constrains what can be done from all possible programs to the set of programs corresponding to the requirements**

Design within the Lifecycle – 3

- Consider the constraints imposed in the software lifecycle
  - » **Putting together a requirements document constrains what can be done from all possible programs to the set of programs corresponding to the requirements**
  - » **The specification formalizes the requirements and in the process adds more constraints.**

Design within the Lifecycle – 4

- Consider the constraints imposed in the software lifecycle
  - » **Putting together a requirements document constrains what can be done from all possible programs to the set of programs corresponding to the requirements**
  - » **The specification formalizes the requirements and in the process adds more constraints.**
    - > **The set of possible programs is smaller**

Design within the Lifecycle – 5

- Consider the constraints imposed in the software lifecycle
  - » **Putting together a requirements document constrains what can be done from all possible programs to the set of programs corresponding to the requirements**
  - » **The specification formalizes the requirements and in the process adds more constraints.**
  - » **Architectural design adds constraints, and so on.**

Design within the Lifecycle – 6

- Consider the constraints imposed in the software lifecycle
  - » **Putting together a requirements document constrains what can be done from all possible programs to the set of programs corresponding to the requirements**
  - » **The specification formalizes the requirements and in the process adds more constraints.**
  - » **Architectural design adds constraints, and so on.**
  - » **Even implementation (programming) adds constraints by specifying in detail every when and what and so is a part of the design process.**

Design within the Lifecycle – 7

- At each stage, there are fewer choices for the **what** and **when** in the final program text

Design within the Lifecycle – 8

- At each stage, there are fewer choices for the **what** and **when** in the final program text
- At each stage the choices must be made within the constraints imposed by the earlier choice

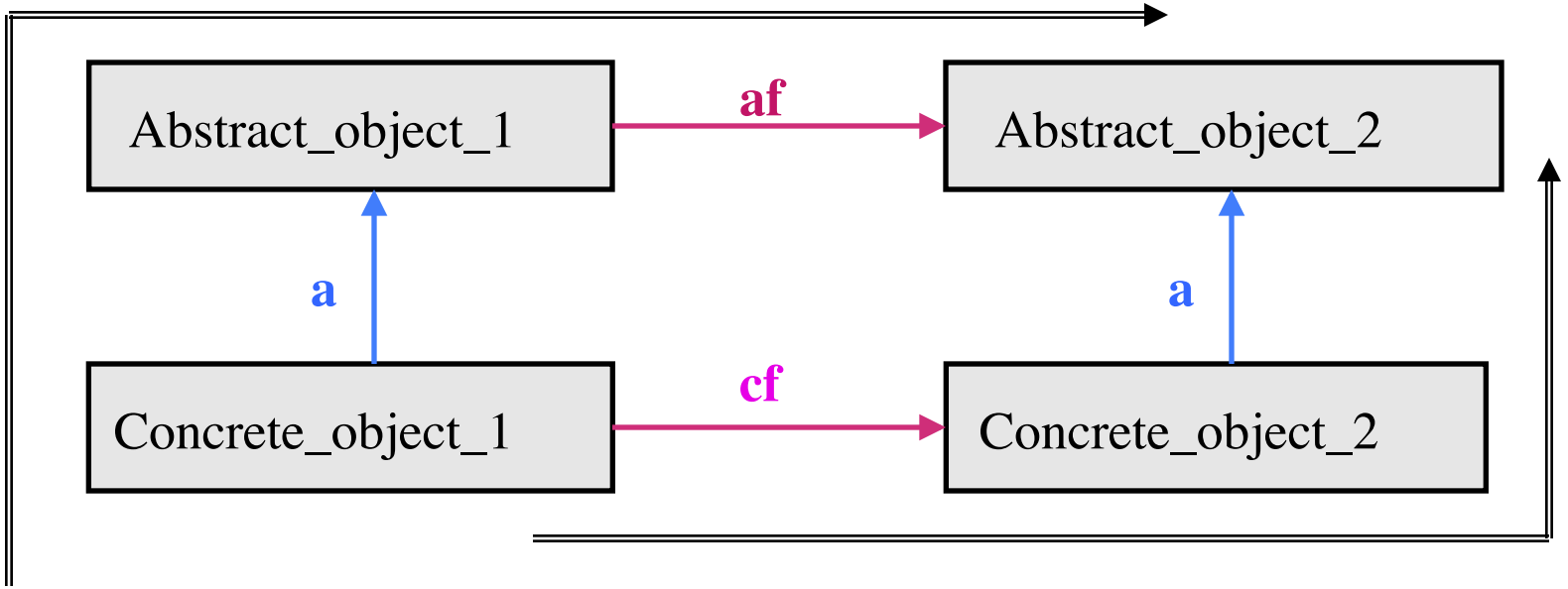
Design within the Lifecycle – 9

- At each stage, there are fewer choices for the **what** and **when** in the final program text
- At each stage the choices must be made within the constraints imposed by the earlier choices
  - » **Or else, backtracking to earlier stages is required**

Design within the Lifecycle – 10

- At each stage, there are fewer choices for the **what** and **when** in the final program text
- At each stage the choices must be made within the constraints imposed by the earlier choices
  - » **Or else, backtracking to earlier stages is required**
- At the end of the implementation stage
  - » **All constraints have been specified, no choices remain, there is the complete program text defining a single executable system**

Class–ADT Consistency Property

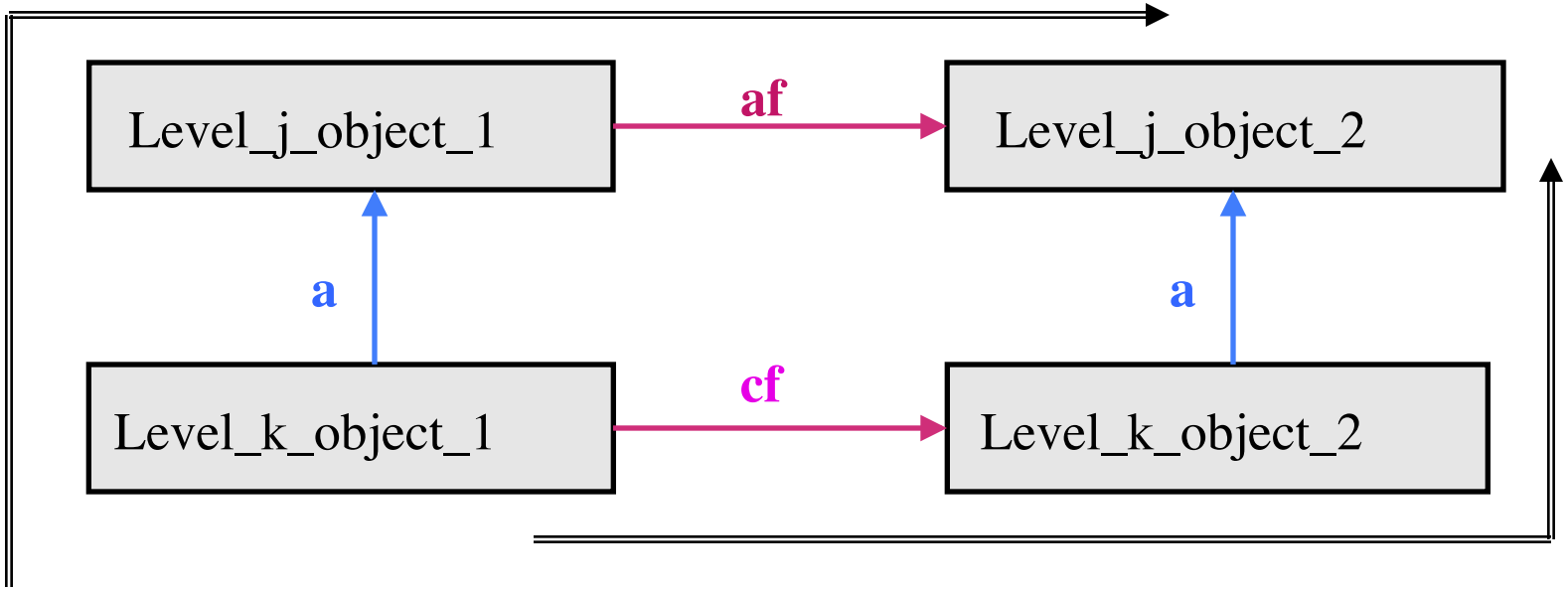


a ; af ≡ cf ; a

Remember this?

Apply to design documentation

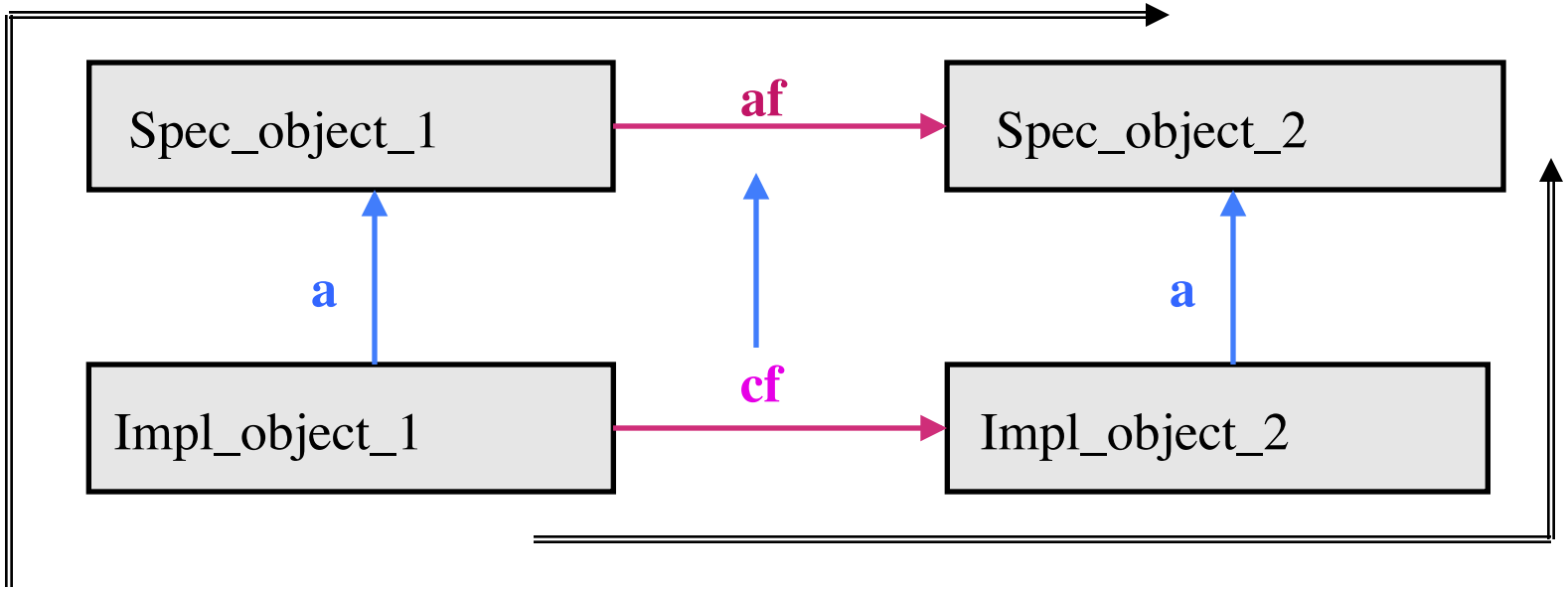
Level j –Level k Consistency Property



a ; af ≡ cf ; a

Levels refer to waterfall levels with j < k

Impl–Spec Consistency Property



a ; af ≡ cf ; a

Document the boxes at the appropriate level

Document correspondence between specification and program text

The blue arrows

Do for every pair of levels

Seamlessness

Since design pervades the entire software lifecycle it is important that supporting methods should apply to the entire lifecycle in a way that minimizes the gaps between successive activities

Seamlessness – 2

Since design pervades the entire software lifecycle it is important that supporting methods should apply to the entire lifecycle in a way that minimizes the gaps between successive activities

Corollary: Should be easy to move information among different notations

formal – program text and mathematics

<—> informal – documentation text

<—> informal – diagrams

Principles of Public Design

Principles of Public Design – 1

- Principle of Use
  - » **Programs will be used by people**

Principles of Public Design – 2

- Principle of Use
  - » **Programs will be used by people**
- Principle of Misuse
  - » **Programs will be misused by people**

Principles of Public Design – 3

- Principle of Use
  - » **Programs will be used by people**
- Principle of Misuse
  - » **Programs will be misused by people**
- Principle of evolution
  - » **Programs will be changed by people**

Principles of Public Design – 4

- Principle of Use
  - » **Programs will be used by people**
- Principle of Misuse
  - » **Programs will be misused by people**
- Principle of evolution
  - » **Programs will be changed by people**
- Principle of migration
  - » **Programs will be moved to new environments by people**

DSQ* – Readable & Understandable

- All Design artifacts – **program text included** – are primarily to be read and used by people.

*DSQ

Design for Software Quality

DSQ* – Readable & Understandable – 2

- All Design artifacts – **program text included** – are primarily to be read and used by people.

- Execution is incidental

DSQ* – Readable & Understandable

- All Design artifacts – **program text included** – are primarily to be read and used by people.

- Execution is incidental

Primary purpose of design is to communicate with other people – even you are somebody else in the future, so you must communicate with yourself

DSQ – Works

- Complete – Correct – Usable

- Efficient as it needs to be
  - » **Speed up where necessary after instrumentation**

DSQ – Adaptable

- All programs evolve over time
- Make plausible modifications easy
  - » **A sign of a good design is it is easy to modify and adapt to changing circumstances**
  - » **Could be used in unexpected ways**

DSQ – On Time & Budget

- Time is money
  - » **Pay back on investment**
- Imbedded systems
  - » **Programs are only a part of the system**
  - » **All systems are part of a larger system**

DSQ – Correctness

- The ability of a system to perform according to specification
  - » **First write correct programs**
    - > **Then worry about efficiency!!!**
  - » **A fast program that is wrong**
    - > **Is worse than useless**

DSQ – Efficiency

- Use an appropriate amount of resources for the task
  - » **Space for storing data and temporary results**
  - » **Execution time**
  - » **Space – time tradeoff**
  - » **Communications bandwidth**

DSQ – Robustness & Ease of Use

- Robustness
  - » **The ability of a software system to react in a reasonable manner to cases not covered by the specification**
    - > **Works correctly for defined inputs**
    - > **Recover gracefully from unexpected inputs**
    - > **Recover gracefully from hardware and algorithm errors**
- Ease of use
  - » **Including installation**

DSQ – Reuse

- Use variations in different software products
  - » **Same as ... except …**
    - > **I.e. make changes, then use**

- The ability of a software system to react in a reasonable manner when reused

DSQ – Reuse – 2

- NOT just using
  - » **A pot is not reused when boiling water**
    - > **It is meant to boil water on many different occasions**
- Reuse
  - » **Pot is used to bail a boat**
    - > **Maybe by bending it to fit the shape of the hull**

Design Principle (DP) – Abstraction

- Extract fundamental parts
  - » **Describe what is wanted**
- Ignore the inessential
  - » **Do not describe what is not wanted**

Design Principle (DP) – Encapsulation

- Information Hiding
  - » **Expose only what the user needs to know**
    - > **The interface**
  - » **Hide implementation details**

DP – Modularity

- Handle complexity
  - » **Use divide and conquer**

- Minimize interaction between parts

OO Design Techniques (OODT)

- Basis consists of
  - » **Classes**
    - > **Define abstract data types**
  - » **Objects**
    - > **Are instances of those types**

OODT Interfaces & Strong Typing

- Interface
  - » **Gives the user what they need to know to use objects from a given class**
    - > **API – Application Program Interface**

- Strong typing
  - » **Enforces objects are used correctly by type**
    - > **Do not take square root of a colour**

OODT Inheritance and Polymorphism

- Inheritance
  - » **Single and multiple**
    - > **Provides for reuse**

- Polymorphism
  - » **Invoke the proper method for an object depending upon its type**

OODT Assertions

- Equip a class and its features with
  - » **Pre and post conditions**
  - » **Class invariants**
  - » **Loop invariants**
- Use tools to produce documentation from these assertions
- Monitor assertions at run time

OODT Information Hiding

- Specify what features are available
  - » **To all clients**
  - » **Some clients**
  - » **No clients**

OODT Exception Handling

- Support robustness with a mechanism too recover from abnormal situations

OODT Genericity

- Genericity
  - » **Write classes with formal generic parameters representing arbitrary types**
- Constrained genericity
  - » **Arises from genericity and inheritance to constrain formal generic parameters to a specific type**

OODT Feature Redefinition

- Reuse requires the ability to modify an object for a new environment so features can be redefined

- Some design decisions must be deferred so provide a means to specify the interface of a feature without defining how it does it.

Structural Design Aspects

- Tokenization
  - » **What kinds of symbols are in the input and output**
- Data structures
  - » **How and what data structures should be selected**
- Program structures
  - » **How should a program be structured**

Structural Design Aspects – 2

- Procedure partitioning
  - » **How should one decide when a set of operations be made into a procedure**
- Class partitioning
  - » **How to decide what goes into a class or module**
- Correspondence
  - » **When do structures correspond**
  - » **When to use communicating sequential processes**