Architectural styles (according to CMU's SEI)

Architecture description based on architectural styles

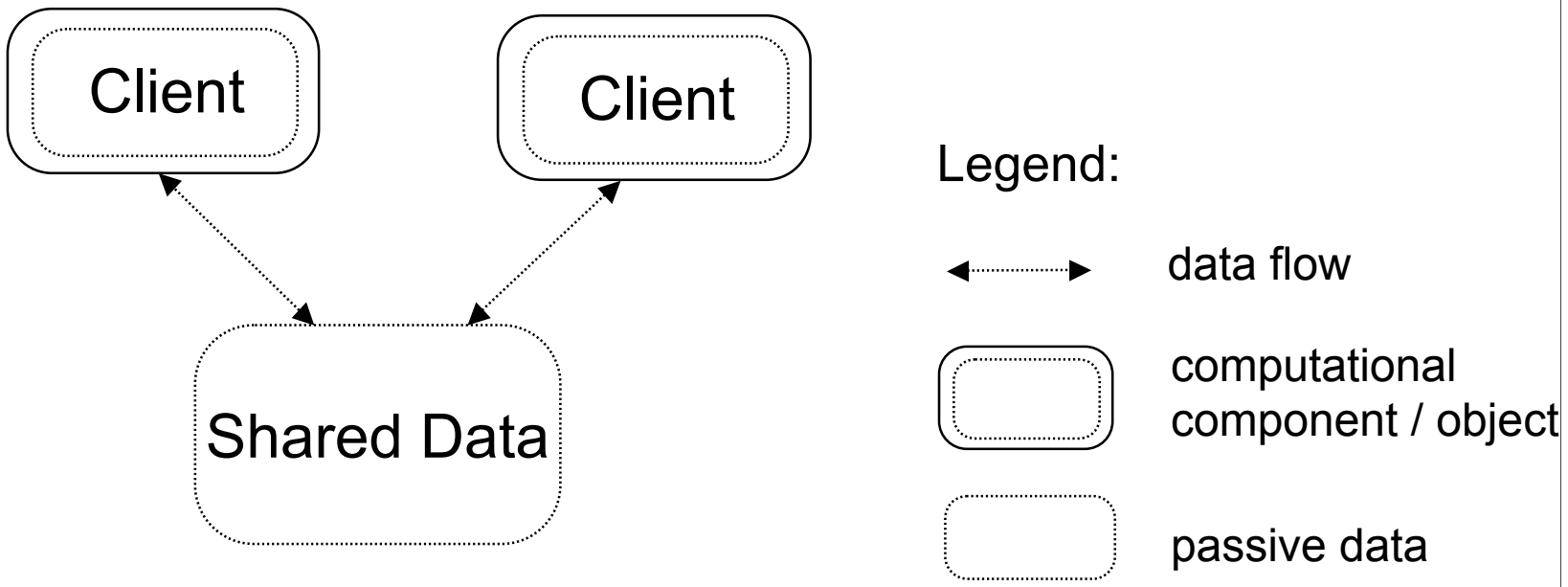
- **Data-centered:**
  - **Repository**
  - **Blackboard**
- **Data-flow:**
  - **Pipes & filters**
  - **Batch/sequential**
- Call-and-return:
  - Top down
  - OO
  - layered

» Virtual machine:
  - Interpreter
  - Rule-based

» Independent components:
  - Communicating processes
  - Event systems
    - implicit invocation
    - explicit invocation

Data-centered (I)

Access to shared data represents the core characteristic of data-centered architectures. The data integrability forms the principal goal of such systems.



Data-centered (II)

The means of communication between the components distinguishes the subtypes of the data-centered architectural style:

- **Repository: passive data** (see schematic representation of previous slide)
- **Blackboard: active data**
  A blackboard sends notification to subscribers when relevant data change (→ Observer pattern)

Data-centered (III)

+ clients are quite independent of each other
  => clients can be modified without affecting others

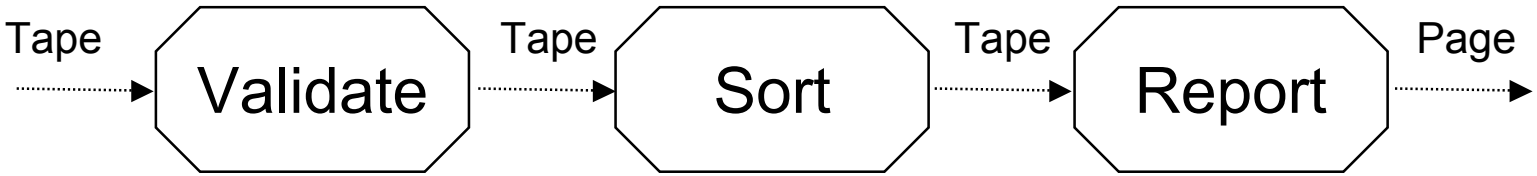
  coupling between clients might increase performance but lessen this benefit

+ new clients can be easily added

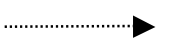
No rigid separation of styles: When clients are independently executing processes: client/server architectural style

Data-flow

The system consists of a **series of transformations on successive pieces of (input) data.** Reuse and modifiability form the principal goals of such architectures.



Legend:


data flow

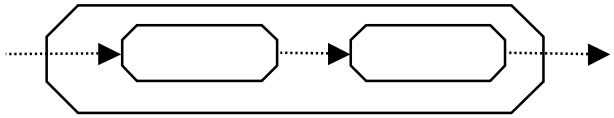

process

Data-flow substyles

- **Batch sequential** (→ sample on previous slide)
  - components (= processing steps) are independent programs
  - **each step runs to completion before the next step starts**, i.e., each batch of data is transmitted as a whole between steps
- **Pipe-and-filter** (→ UNIX pipes & filters)
  - **incremental transformation of data** based on streams
  - filters are stream transducers and use little contextual information and retain no state information between instantiations
  - pipes are stateless and just move data between filters

Pros and cons of pipes-and-filters

+ no complex component interactions to manage
+ filters are black boxes
+ pipes and filters can be hierarchically composed



– batch mentality => hardly suitable for interactive applications
– filter ordering can be difficult; filters cannot interact cooperatively to solve a problem
– performance is often poor

  parsing/unparsing overhead due to lowest common denominator data representation
– filters which require all input for output production have to create unlimited buffers

Virtual machine (I)

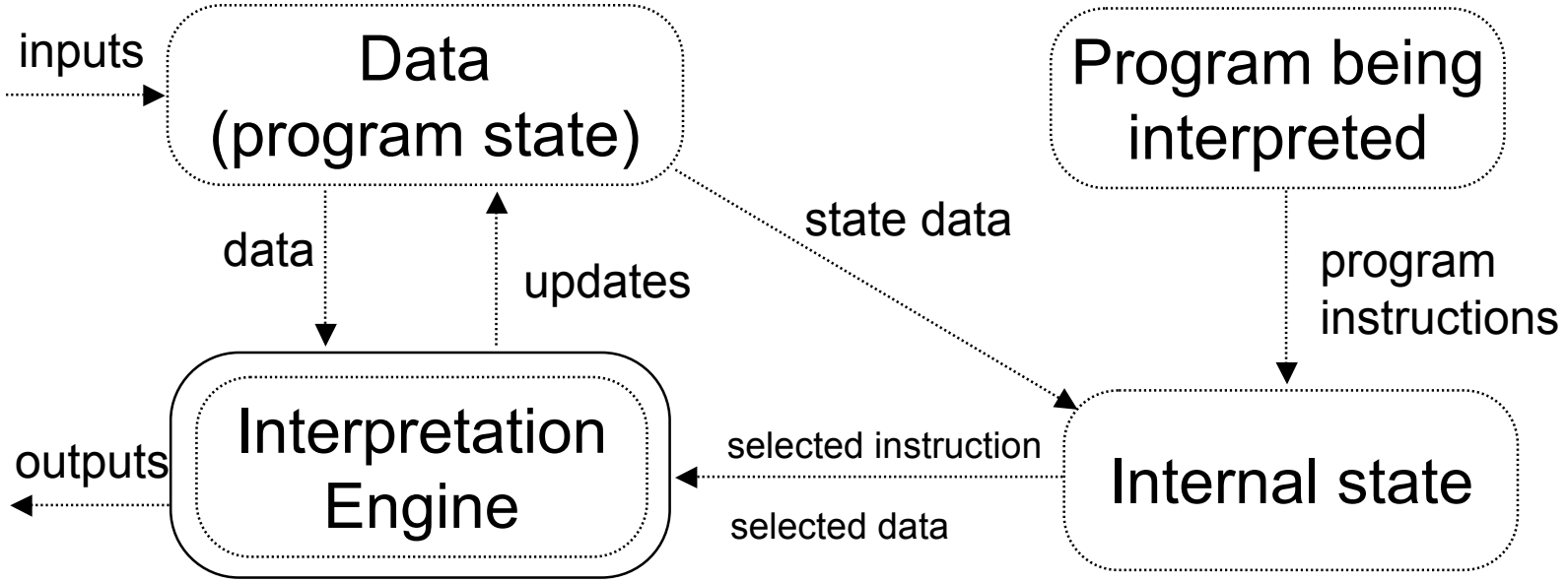
Virtual machines **simulate some functionality that is not native to the hardware/software on which it is implemented.** This supports achieving the quality attribute of **portability**.

Examples:

- interpreters
- command language processors
- rule-based systems

Virtual machine (II)

Schematic representation:



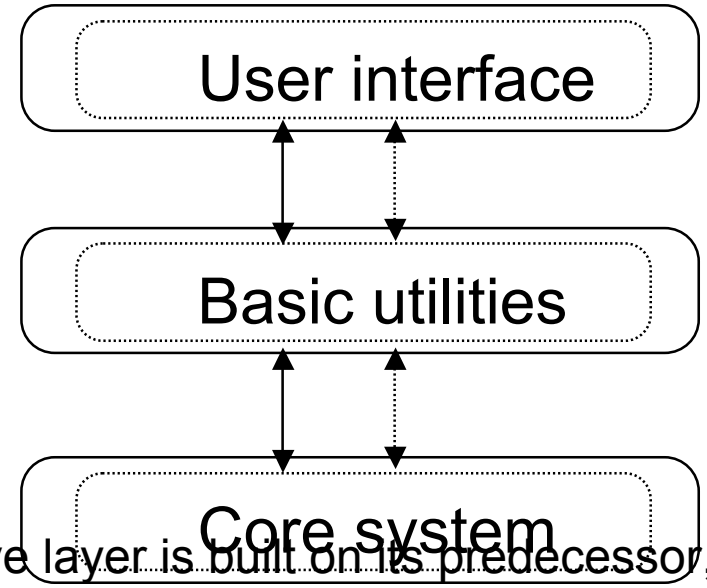
Call-and-return

Call-and-return architectures rely on the well-known abstraction of procedures/functions/methods. Shaw and Garlan discern between the following substyles:

- main-program-and-subroutine style
  - remote-procedure-call systems also belong to this category but are decomposed in parts that live on computers connected via a network
- object-oriented or abstract-data-type style
- layered style

Layered style

Components belong to layers. In pure layered systems **each level should communicate only with its immediate neighbors.**

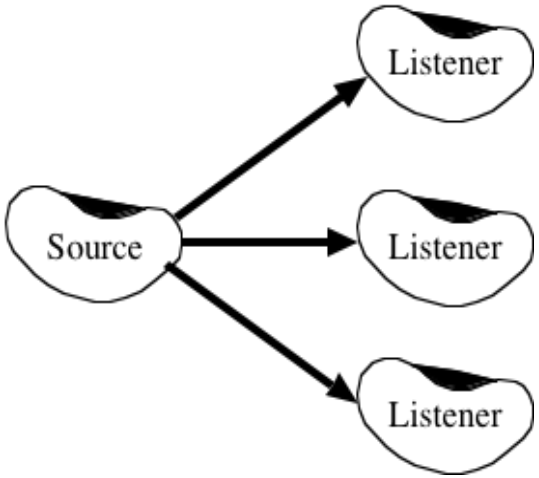


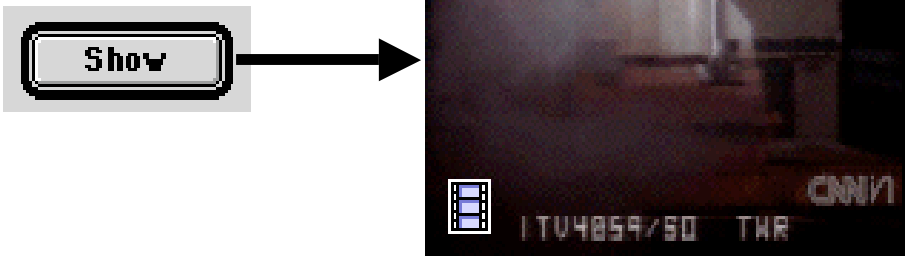
Each successive layer is built on its predecessor, hiding the lower layer and providing some services that the upper layers make use of. Upper layers often form virtual machines.

Event systems

Publish/subscribe (observer) pattern: Components can register an interest in notifications.

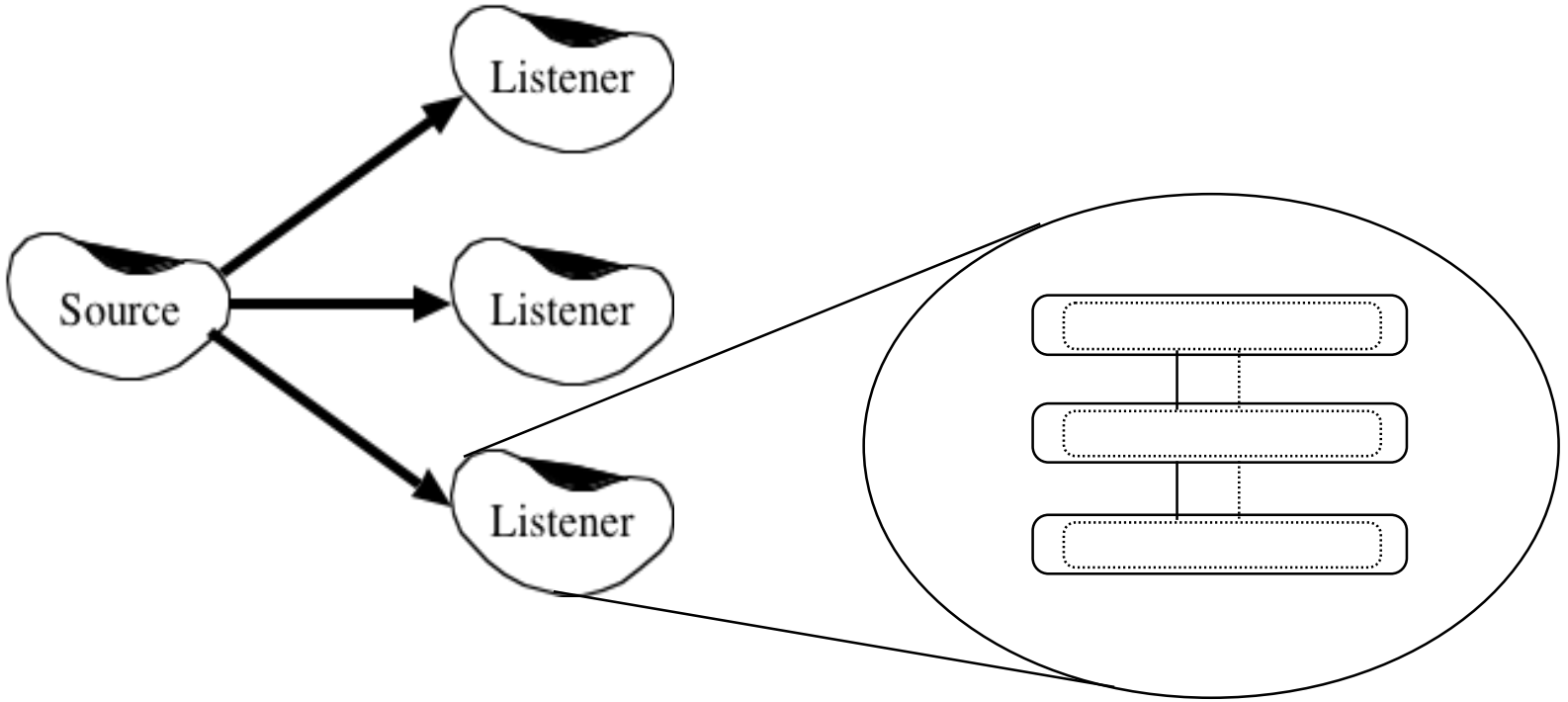
Example: coupling between JavaBeans





Heterogeneous styles (I)

Example: event system + layered style



Heterogeneous styles (II)

In general, the presented architectural styles do not clearly categorize architectures. Styles exist as cognitive aids and communication cues.

- The data-centered style, composed out of thread-independent clients is like an independent component architecture.
- The layers in a layered architecture might be objects/ADTs.
- The components in a pipe-and-filter architecture are usually independently operating processes and thus also correspond to an independent component architecture.
- Commercial client/server systems with a CORBA-based infrastructure could be described as layered object-based process systems, i.e., a hybrid of three styles.

Architecture description

Architectural styles are often insufficient to describe a system‘s architecture

Architectural styles do not clearly categorize architectures. Thus they do not suffice to describe architectures as a whole.

Consider the following sample cases:

- The layers in a layered architecture might be objects/ADTs.
- Commercial client/server systems with a CORBA-based infrastructure could be described as **layered object-based process systems**, i.e., a hybrid of three styles.

Choose an appropriate mix of various notations + informal description

As a consequence, one has to decide on the appropriate description, which will be a mix of the following principal options:

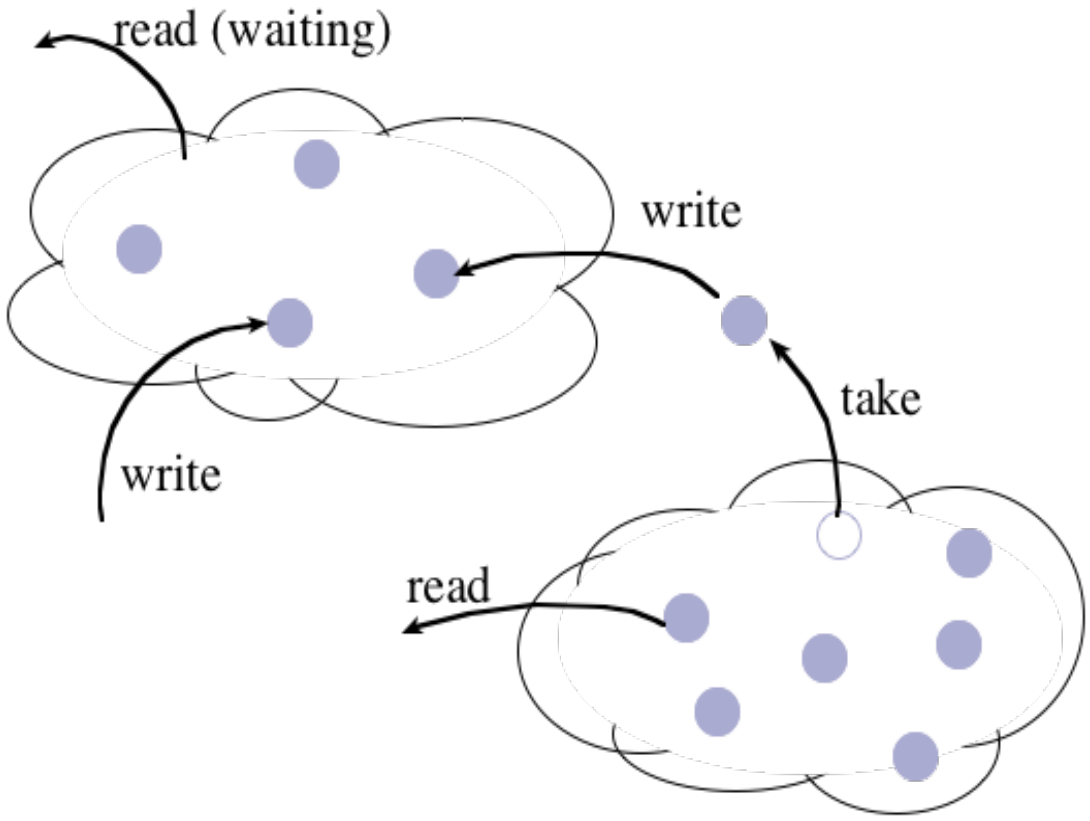
- schematic representation according to CMU/SEI
- 4+1 View Model of Architecture (Kruchten , 1995)
- UML:
  - **subsystem/package-diagrams**
  - **class-diagrams**
  - **interaction-/object-diagrams**
  - **state-diagrams**
- any schematic figures that help; informal text as glue
- source code fragments of coarse-grained components or component interfaces
- formal specifications, eg, with architecture description languages

Sample architectural description of JavaSpaces

JavaSpaces characteristics (from an architectural point of view):

- data-centered, mainly a repository architectural style, sometimes black-board architectural style
- main design goals
  - **extensibility** through loose coupling of distributed processes and distributed Java components
  - **simplicity** from a reuser‘s point of view

JavaSpaces architecture overview



Characteristics of JavaSpaces (JS)

- **high-level coordination tool** for gluing processes and components together in a distributed system **without message passing and remote method invocation**
- a **space is a shared, network-accessible repository for objects**
- instead of communicating directly, JS apps consist of processes that coordinate by exchanging objects through spaces

Sample JS use scenarios (I)

Example isn‘t another way to teach, it is the only way to teach (Albert Einstein)

(1) Space acting as “auction room“:

Sellers deposit for-sale items with descriptions and asking prices as objects into the space.

Buyers monitor the space for items that interest them. If an item interest them they put bid objects into the space.

Sellers monitor the space for bids.

etc.

Sample JS use scenarios (II)

(2) Compute-intensive jobs

A series of tasks—for example, rendering a frame in a computer animation represents a task—are written into a space.

Participating graphic workstations search the space for rendering tasks. Each one finding tasks to be done, removes it, accomplishes it, writes the result back into the space.

Interfaces of JS key abstractions and their usage

The **rest of the architecture description could be** a series of documented **source code** and **commented UML class and interaction diagrams** that illustrate the simplicity of reusing the JS architecture.

```
public calls SampleMsg implements Entry {  // empty JS interface
        ...
}

// putting an object into a space
SampleMsg msg= new SampleMsg();
JavaSpace space= SpaceAccessor.getSpace();
space.write(msg, ...); // other parameters omitted
...
```

Architecture analysis: The SAAM

When and Why To Analyze Architecture -1

- Analyzing for system qualities early in the life cycle allows for a comparison of architectural options.
- When building a system
  - Architecture is the earliest artifact where trade-offs are visible.
  - Analysis should be done when deciding on architecture.
  - The reality is that analysis is often done during damage control, later in the project.

When and Why To Analyze Architecture -2.

- When acquiring a system
  - Architectural analysis is useful if the system will have a long lifetime within organization.
  - Analysis provides a mechanism for understanding how the system will evolve.
  - Analysis can also provide insight into other visible qualities.

Qualities Are Too Vague for Analysis

- Is the following system modifiable?
  - Background color of the user interface is changed merely by modifying a resource file.
  - Dozens of components must be changed to accommodate a new data file format.
- A reasonable answer is
  - yes with respect to changing background color
  - no with respect to changing file format

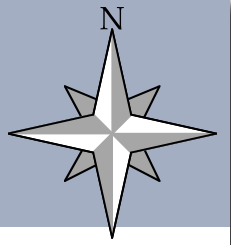
Qualities Are Too Vague for Analysis

- Qualities only have meaning within a context.
- SAAM specifies context through scenarios.

Scenarios

- A scenario is a brief description of a stakeholder's interaction with a system.
- When creating scenarios, it is important to consider all stakeholders, especially
  - end users
  - developers
  - maintainers
  - system administrators

Steps of a SAAM Evaluation



- Identify and assemble stakeholders
- Develop and prioritize scenarios
- Describe candidate architecture(s)
- Classify scenarios as direct or indirect
- Perform scenario evaluation
- Reveal scenario interactions
- Generate overall evaluation

Step 1: Identify and Assemble Stakeholders -1

Stakeholder	Interest
Customer	Schedule and budget; usefulness of system; meeting customers' (or market's) expectations
End user	Functionality, usability
Developer	Clarity and completeness of architecture; high cohesion and limited coupling of parts; clear interaction mechanisms
Maintainer	Maintainability; ability to locate places of change

Step 1: Identify and Assemble Stakeholders -2

Stakeholder	Interest
System administrator	Ease in finding sources of operational problems
Network administrator	Network performance, predictability
Integrator	Clarity and completeness of architecture; high cohesion and limited coupling of parts; clear interaction mechanisms

Step 1: Identify and Assemble Stakeholders -3.

Stakeholder	Interest
Tester	Integrated, consistent error-handling; limited component coupling; high component cohesion; conceptual integrity
Application builder (if product line architecture)	Architectural clarity, completeness; interaction mechanisms; simple tailoring mechanisms
Representative of the domain	Interoperability

Step 2: Stakeholders Develop and Prioritize Scenarios

- Scenarios should be typical of the kinds of evolution that the system must support:
  - functionality
  - development activities
  - change activities
- Scenarios also can be chosen to give insight into the system structure.
- Scenarios should represent tasks relevant to all stakeholders.
- Rule of thumb: 10-15 prioritized scenarios

Step 3: Describe Candidate Architectures

- It is frequently necessary to elicit appropriate architectural descriptions.
- Structures chosen to describe the architecture will depend on the type of qualities to be evaluated.
- Code and functional structures are primarily used to evaluate modification scenarios.

Step 4: Classify Scenarios

- There are two classes of scenarios.
  - Direct scenarios are those that can be executed by the system without modification.
  - Indirect scenarios are those that require modifications to the system.
- The classification depends upon both the scenario and the architecture.
- For indirect scenarios we gauge the order of difficulty of each change: e.g. a person-day, person-week, person-month, person-year.

Step 5: Perform Scenario Evaluation

- For each indirect scenario
  - identify the components, data connections, control connections, and interfaces that must be added, deleted, or modified
  - estimate the difficulty of modification
- Difficulty of modification is elicited from the architect and is based on the number of components to be modified and the effect of the modifications.
- A monolithic system will score well on this step, but not on next step.

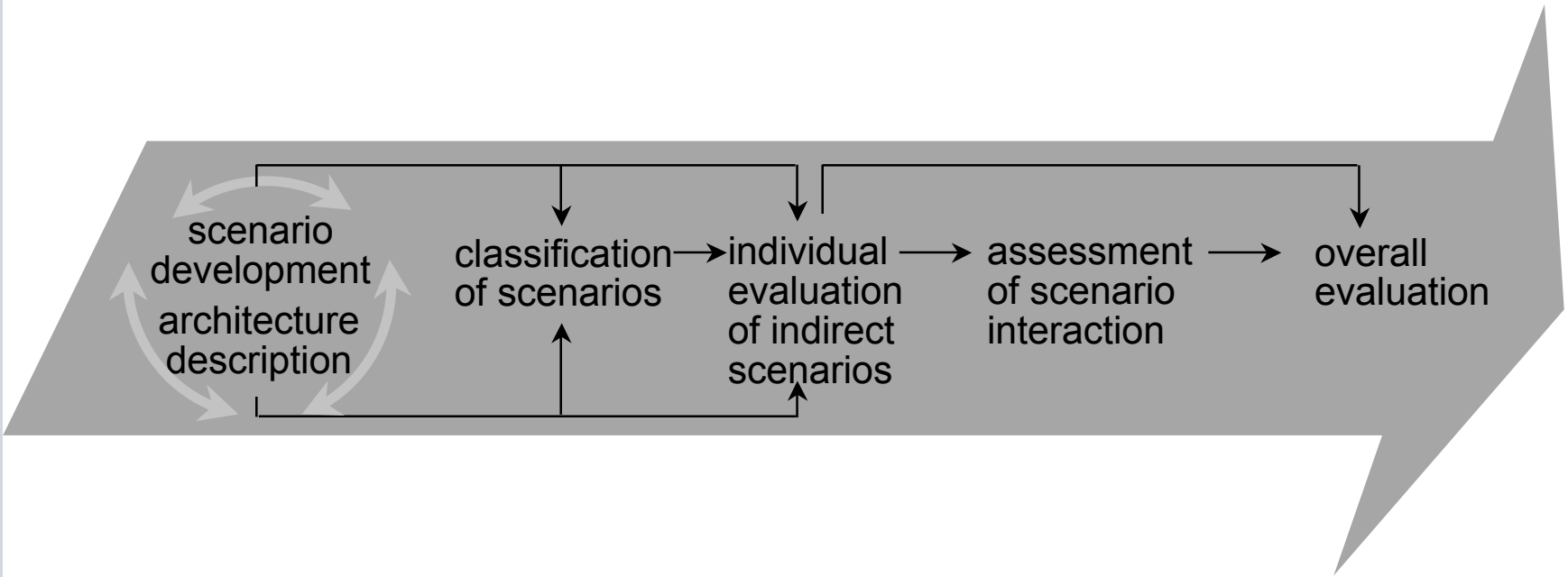
Step 6: Reveal Scenario Interactions

- When multiple indirect scenarios affect the same components, this could indicate a problem.
  - could be good, if scenarios are variants of each other
    - change background color to green
    - change background color to red
  - could be bad, indicating a potentially poor separation of concerns
    - change background color to red
    - port system to a different platform

Step 7: Generate Overall Evaluation

- Not all scenarios are equal.
- The organization must determine which scenarios are most important.
- Then the organization must decide as to whether the design is acceptable “as is” or if it must be modified.

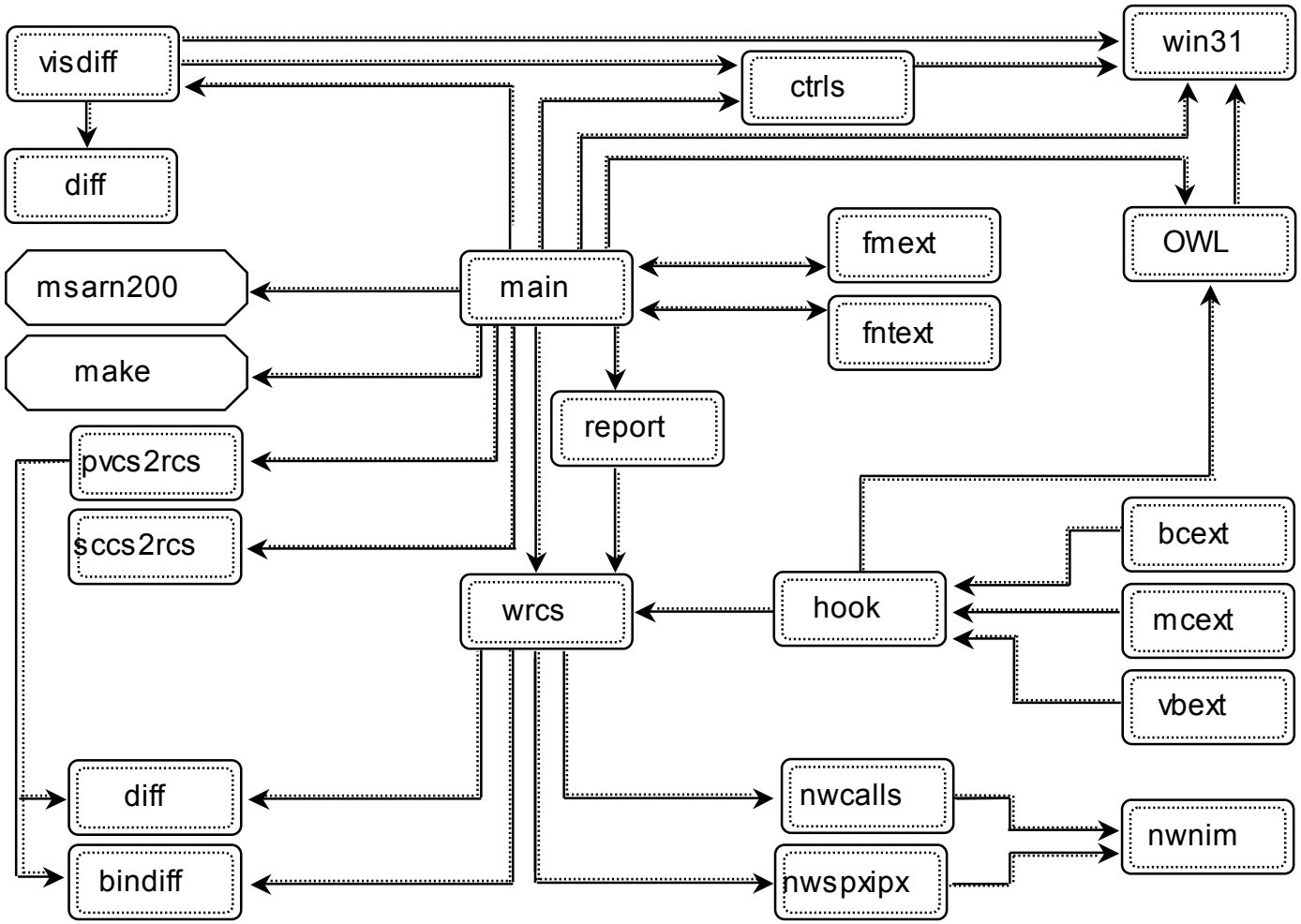
Interaction of SAAM Steps



Example: SAAM Applied to Revision Control System

- "WRCS" is a large, commercially-available revision control system.
- No documented system architecture existed prior to the evaluation.
- The purpose of the evaluation was to assess the impact of anticipated future changes.
- Three iterations were required to develop a satisfactory representation, alternating between
  - development of scenarios
  - representation of architecture

Architectural Representation of WRCS



Scenarios Used in WRCS

- User scenarios
  - compare binary file representations
  - configure the product's toolbar
- Maintainer
  - port to another operating system
  - make minor modifications to the user interface
- Administrator
  - change access permissions for a project
  - integrate with a new development environment

Scenario Classification

- User scenarios
  - compare binary file representations: *indirect*
  - configure the product's toolbar: *direct*
- Maintainer
  - port to another operating system: *indirect*
  - make minor modifications to the user interface: *indirect*
- Administrator
  - change access permissions for a project: *direct*
  - integrate with a new development environment: *indirect*

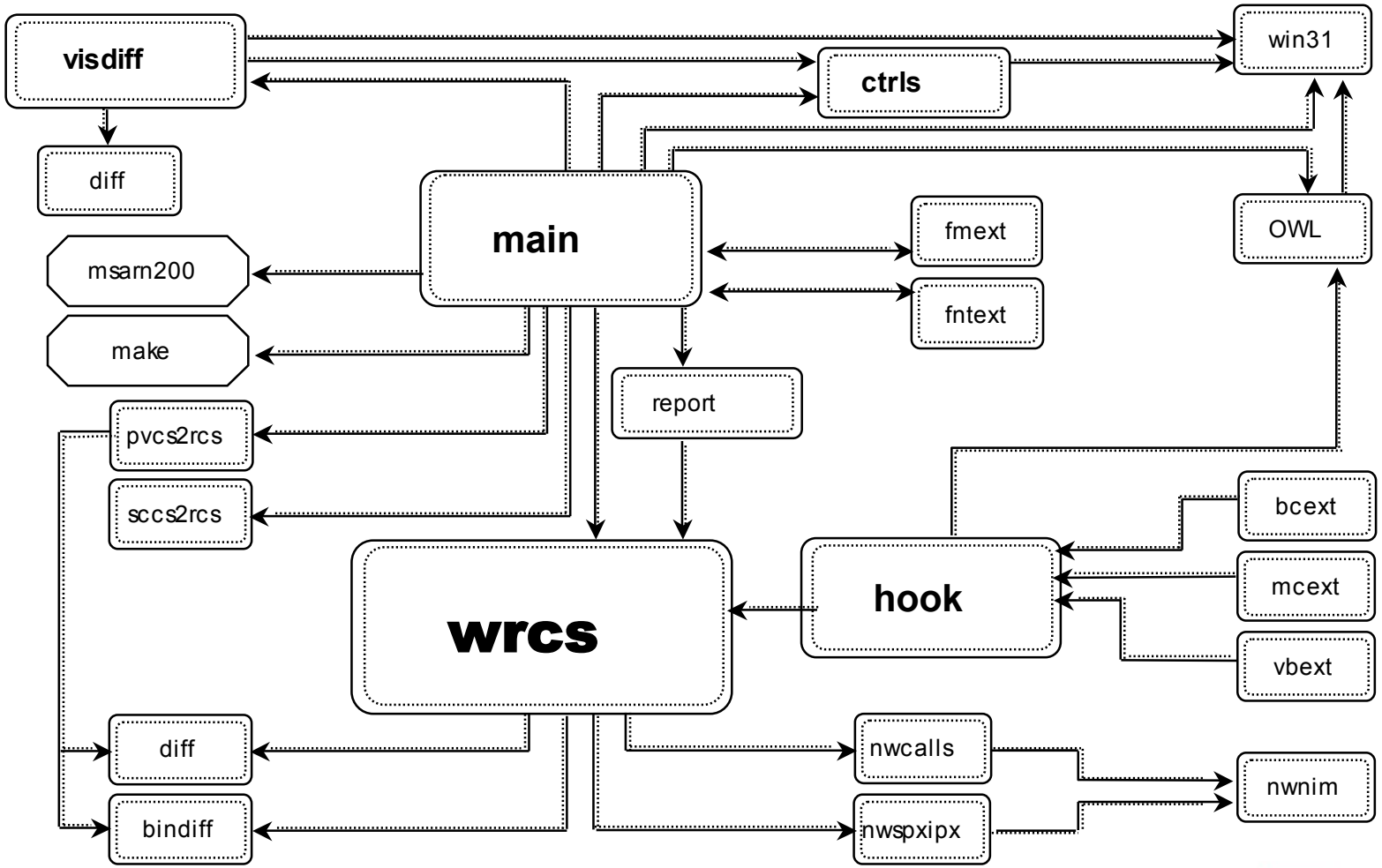
Scenario Interactions

- Each indirect scenario necessitated a change in some modules. This can be represented either tabularly or visually.
- The number of scenarios that affected each module can be shown with a table or graphically, with a fish-eye view.
- A fish-eye view uses size to represent areas of interest.

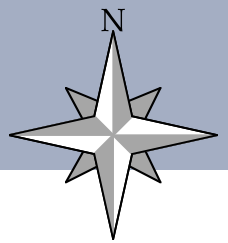
Scenario Interaction Table

Module	No. changes
main	4
wrcs	7
diff	1
bindiff	1
pvcs2rcs	1
sccs2rcs	1
nwcalls	1
nwspxipx	1
nwnlm	1
hook	4
report	1
visdiff	3
ctrls	2

Scenario Interaction Fish-Eye



Lessons Learned from WRCS

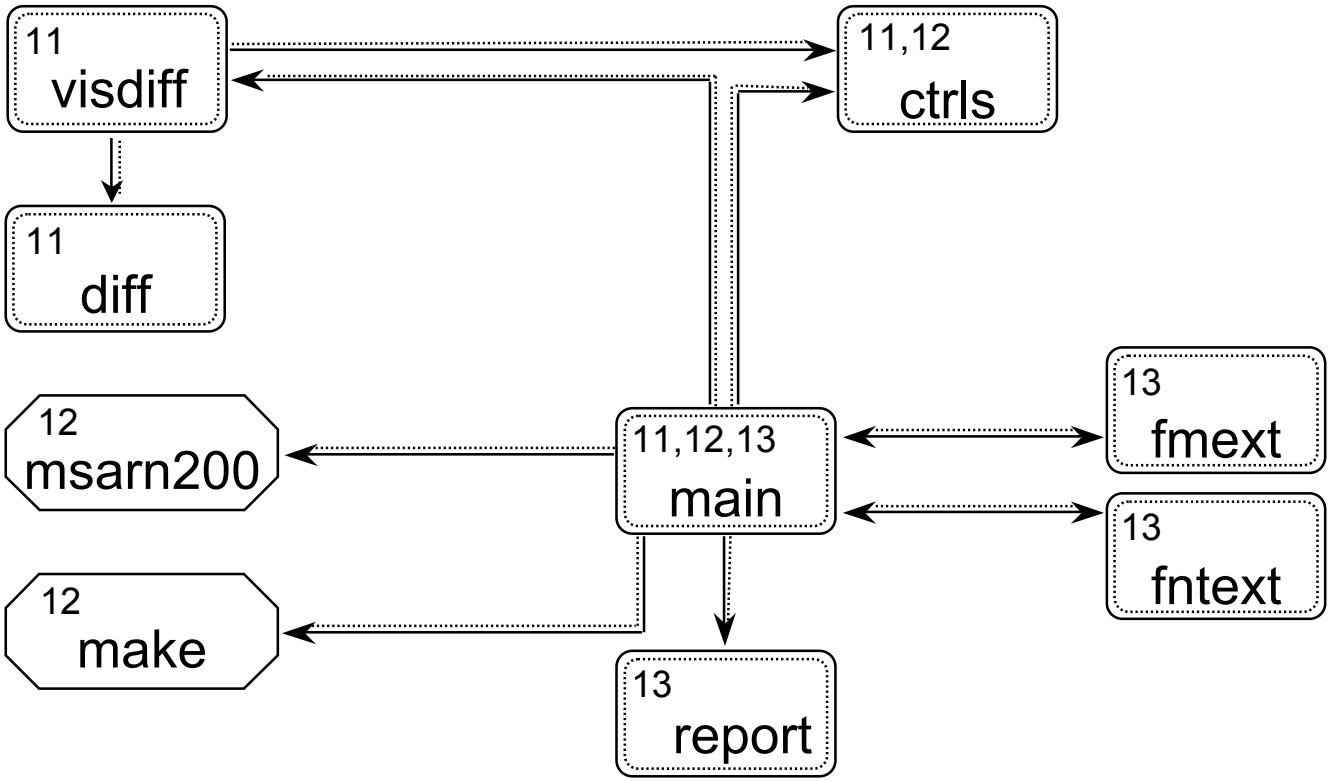


- Granularity of architectural description
- Interpretation of scenario interactions

Proper Granularity of Architectural Description

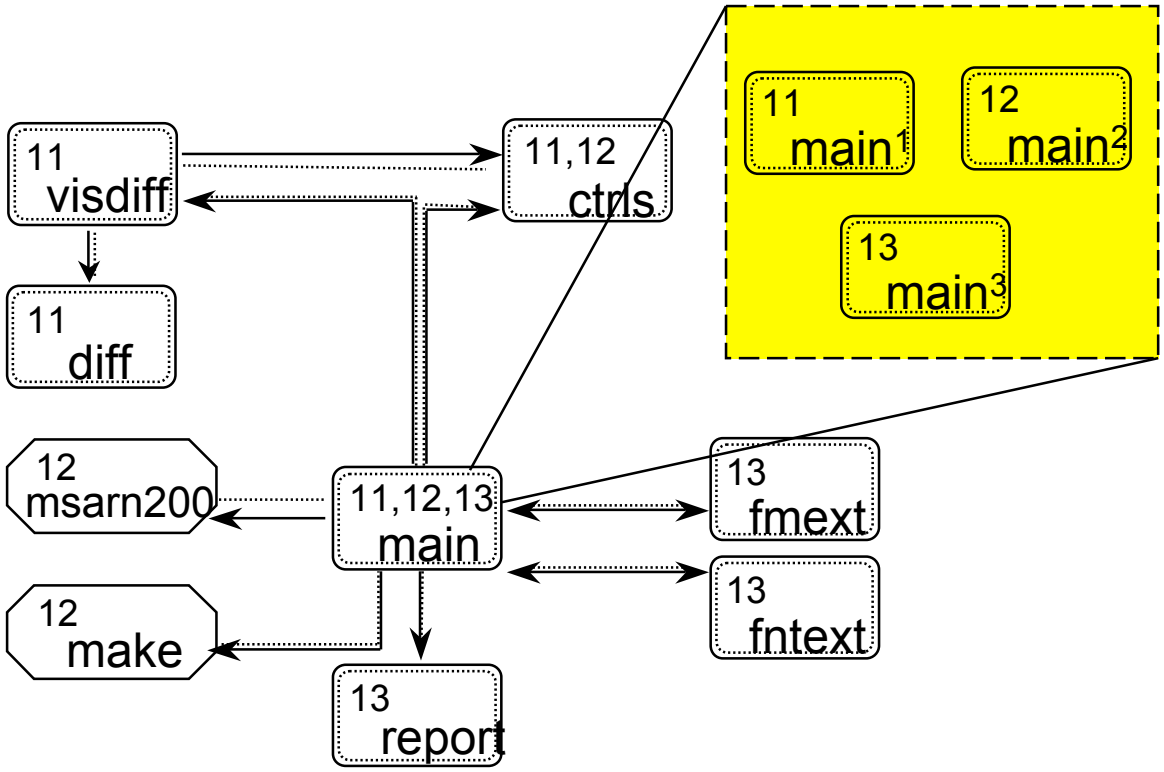
- The level of detail of architectural description is determined by the scenarios chosen.
- The next slide shows what an architect thought was an appropriate level of detail.
- Components are annotated with the numbers of indirect scenarios that affect them.

Original Representation of WRCS



The “main” Scenario Interactions

- Possibilities:
  - Scenarios are all of the same class.
  - Scenarios are of different classes and “main” cannot be subdivided.
  - Scenarios are of different classes, and “main” *can* be subdivided.



WRCS: What did we learn?

- We identified severe limitations in achieving the desired portability and modifiability. A major system redesign was recommended.
- The evaluation itself obtained mixed results.
  - Senior developers/managers found it important and useful.
  - Developers regarded this as just an academic exercise.
- SAAM allowed insight into capabilities and limitations that weren't easily achieved otherwise.
- This was accomplished with only scant knowledge of the internal workings of WRCS.

Lessons from SAAM -1

- Direct scenarios provide a
  - first-order differentiation mechanism for competing architectures
  - mechanism for eliciting and understanding structures of architectures (both static and dynamic)
- It is important to have stakeholders present at evaluation meetings.
  - Stakeholders find it to be educational.
  - Architectural evaluators may not have the experience to keep presenters “honest.”

Lessons from SAAM -2.

- SAAM and traditional architectural metrics
  - Coupling and cohesion metrics do not represent different patterns of use.
  - High scenario interaction shows low cohesion.
  - A scenario with widespread hits shows high coupling.
  - Both are tied to the context of use.
  - SAAM provides a means of sharpening the use of coupling and cohesion metrics.

Summary

- A SAAM evaluation produces
  - technical results: provides insight into system capabilities
  - social results
    - forces some documentation of architecture
    - acts as communication vehicle among stakeholders