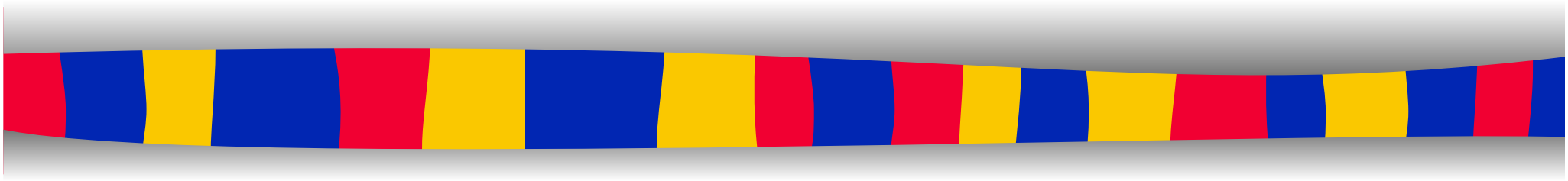
CISC 326

Game Architecture



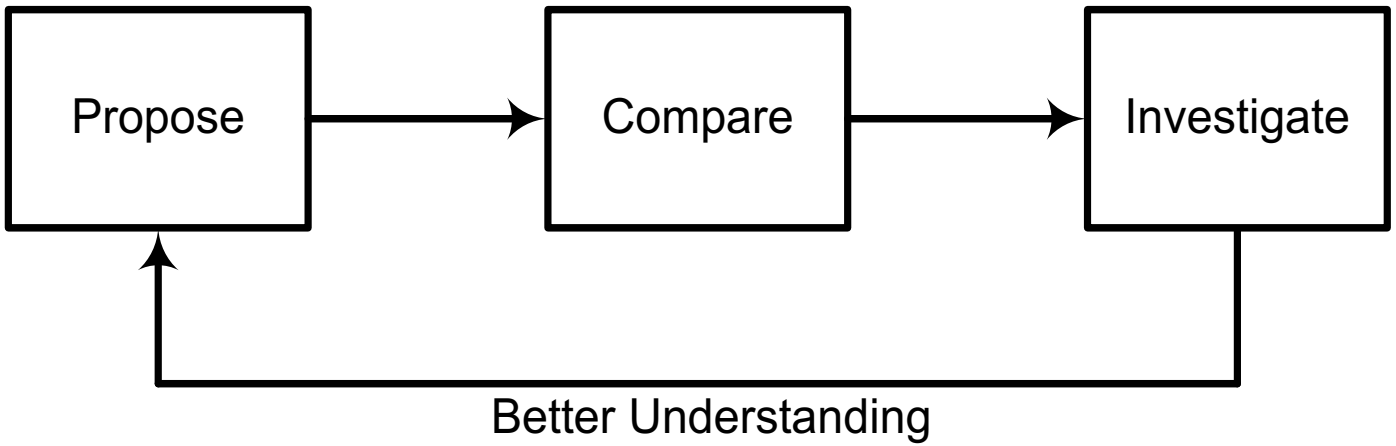
Module 7: Reflexion Models

Ahmed E. Hassan

Understanding Large Systems

- You are asked to provide an estimate on the time needed to implement a particular feature
  - The software system is large
  - Your knowledge of the system is limited
  - Your estimate should be sufficiently accurate

Architecture Understanding Process



- *Propose* conceptual architecture
- *Compare* conceptual with concrete architecture
- *Investigate* gaps

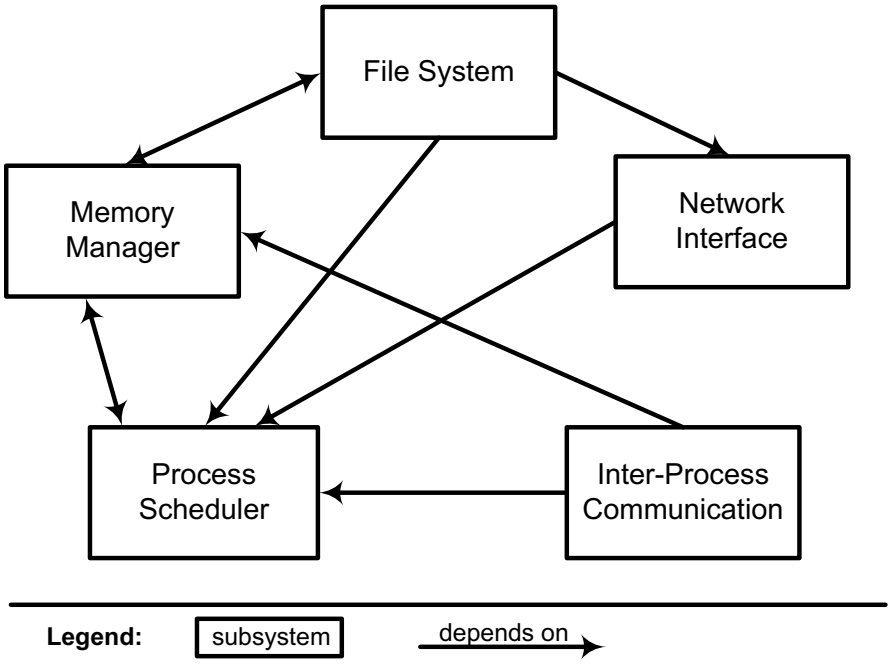
Conceptual Architecture

- Developers propose a conceptual architecture using assumptions and preconceived ideas about the system and its interactions based on:
  - System documentation
  - Developer experience with similar systems
  - Reference architecture
  - Talking to senior developers and domain experts

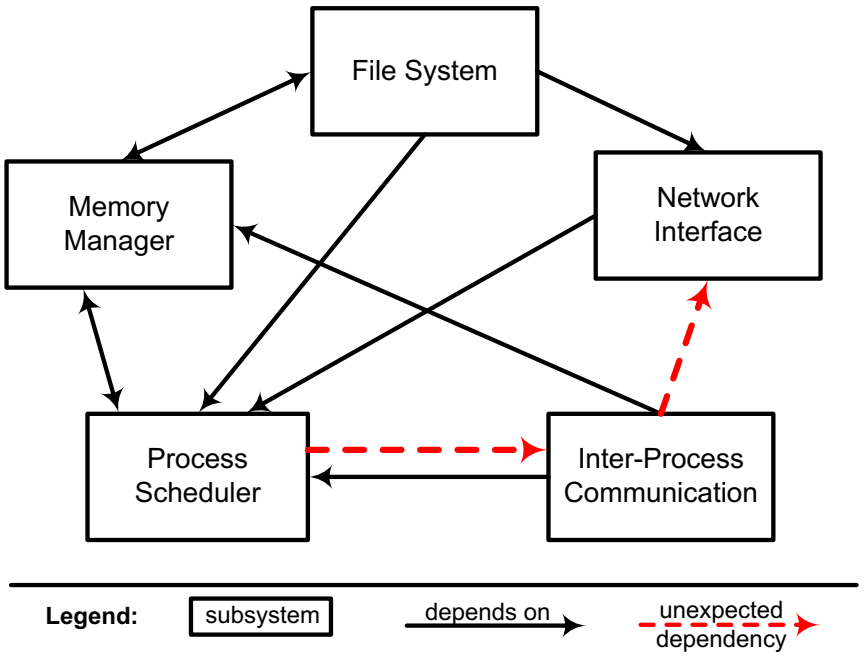
Working on an Operating System

- A developer working on enhancing features in an OS, might being with a conceptual breakdown which consists of five conceptual subsystems:
  - *File System*, *Memory Manager*, *Network Interface*, *Process Scheduler*, and an *Inter-Process Communication*.
- The developer might also assume that these subsystems interact in a particular fashion to implement specific features:
  - *File System* depends on the *Network Interface* to support networked file systems such as NFS.
  - *Memory Manager* depends on the *File System* to support swapping of processes to disk when the system runs out of physical memory.

Operating System Architecture



Conceptual
(*proposed*)

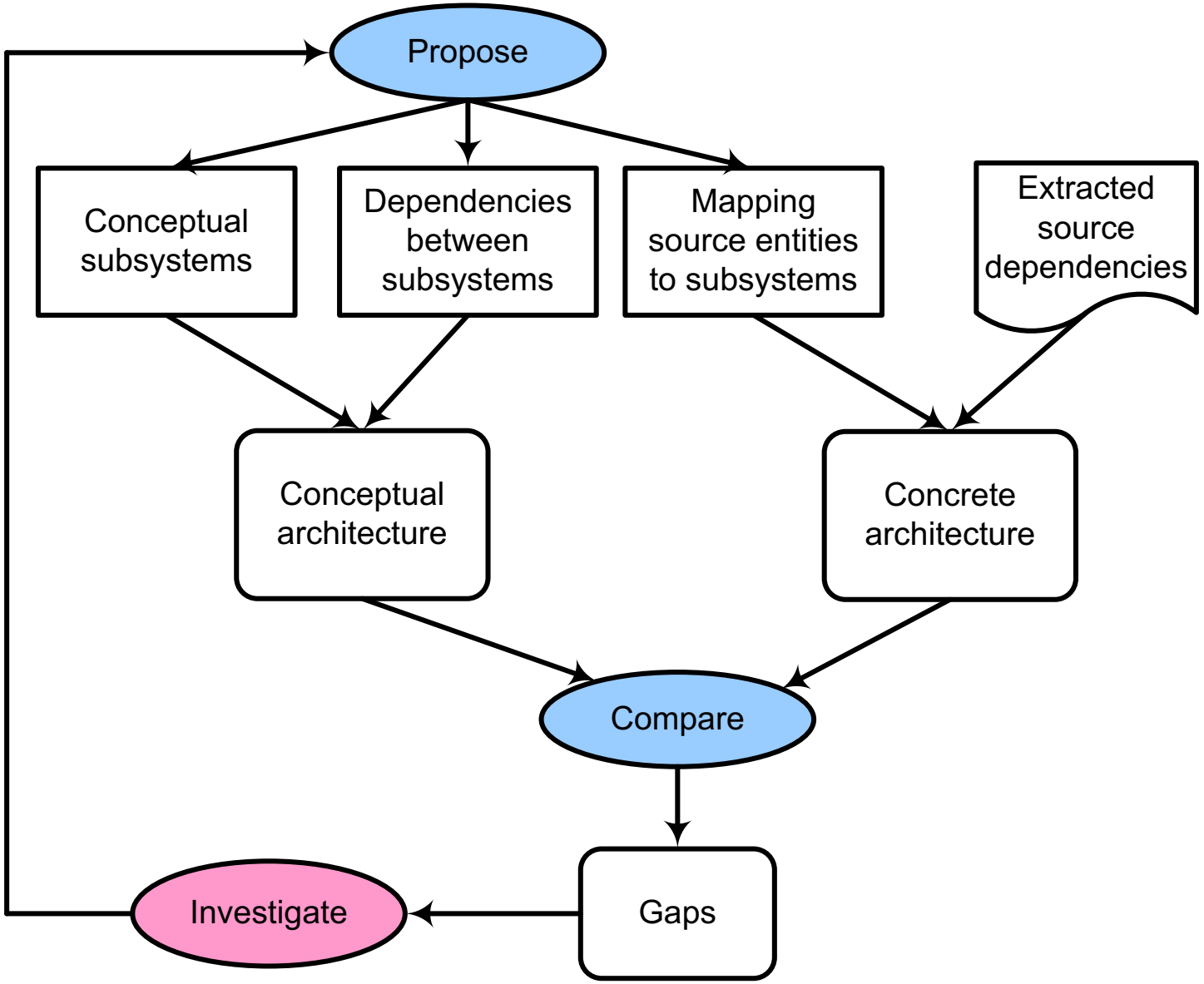


Concrete
(*reality*)

Uncovering the Rationale for the Differences

- Uncovering the rationale is challenging
  - A senior developer
    - may be too busy
    - may not recall the rationale for such dependency
    - may no longer work on the software system
  - The software
    - may have been bought from another company
    - may have its maintenance out-sourced
- Developers must spend hours/days to uncover the rationale. The rationale may be:
  - Justified due to, e.g., optimizations or code reuse; or
  - Not justified due to, e.g., developer ignorance or pressure to market.

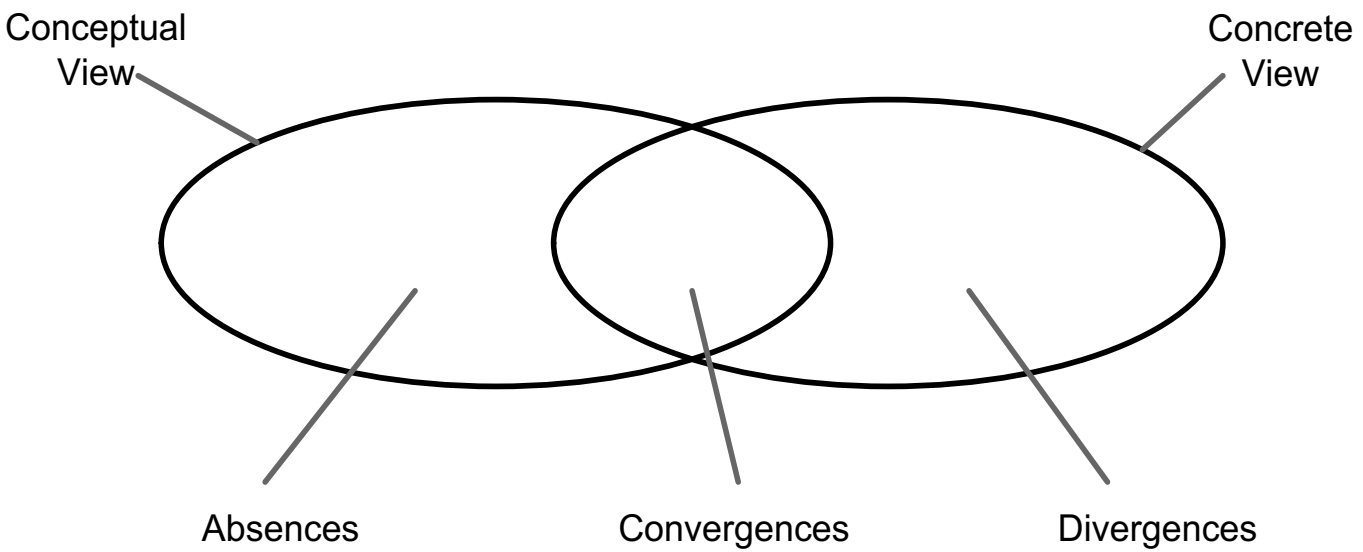
Software Reflexion Framework



Mapping source entities to subsystems

- Mapping files/functions:
  - All files in the “src\sched” directory may be mapped to the *Process Scheduler* subsystem
  - All files in the “src\ipc” directory may be mapped to the *Inter-Process Communication* subsystem
- Mapping dependencies:
  - if a file in “src\ipc“ calls a function defined in another file in “src\sched“ then this is considered to be a dependency relation between the *Inter-Process Communication* and *Process Scheduler* subsystems.

Investigating Gaps



- ***Absences:*** rarely occur in large systems
- ***Convergences:*** usually not a concern
- ***Divergences:*** must investigate dependencies

Which?

- Which concrete source code entities are responsible for an unexpected dependency?
- Based on entity names, we may be able to deduce the reason for the existence of dependencies
  - Names may not help (too cryptic), thus developers find themselves asking several other questions

Who?

- Who introduced an unexpected dependency or removed a missing dependency?
- A knowledge of this person may assist in understanding the reasons for gaps.
- A gap due to a change made by
  - a novice developer may suggest that the developer is at fault and the change must be fixed
  - a senior developer with a well established record for producing high quality code may suggest that the change is correct
- If the change is correct then we may consider updating our conceptual view of the system

When?

- When was the unexpected dependency added or the missing dependency removed?
- Was a change introduced by a senior developer to fix a critical bug under a tight release schedule?
  - E.g. a few days/hours before a release
- Or is it is a justified dependency that we should expect

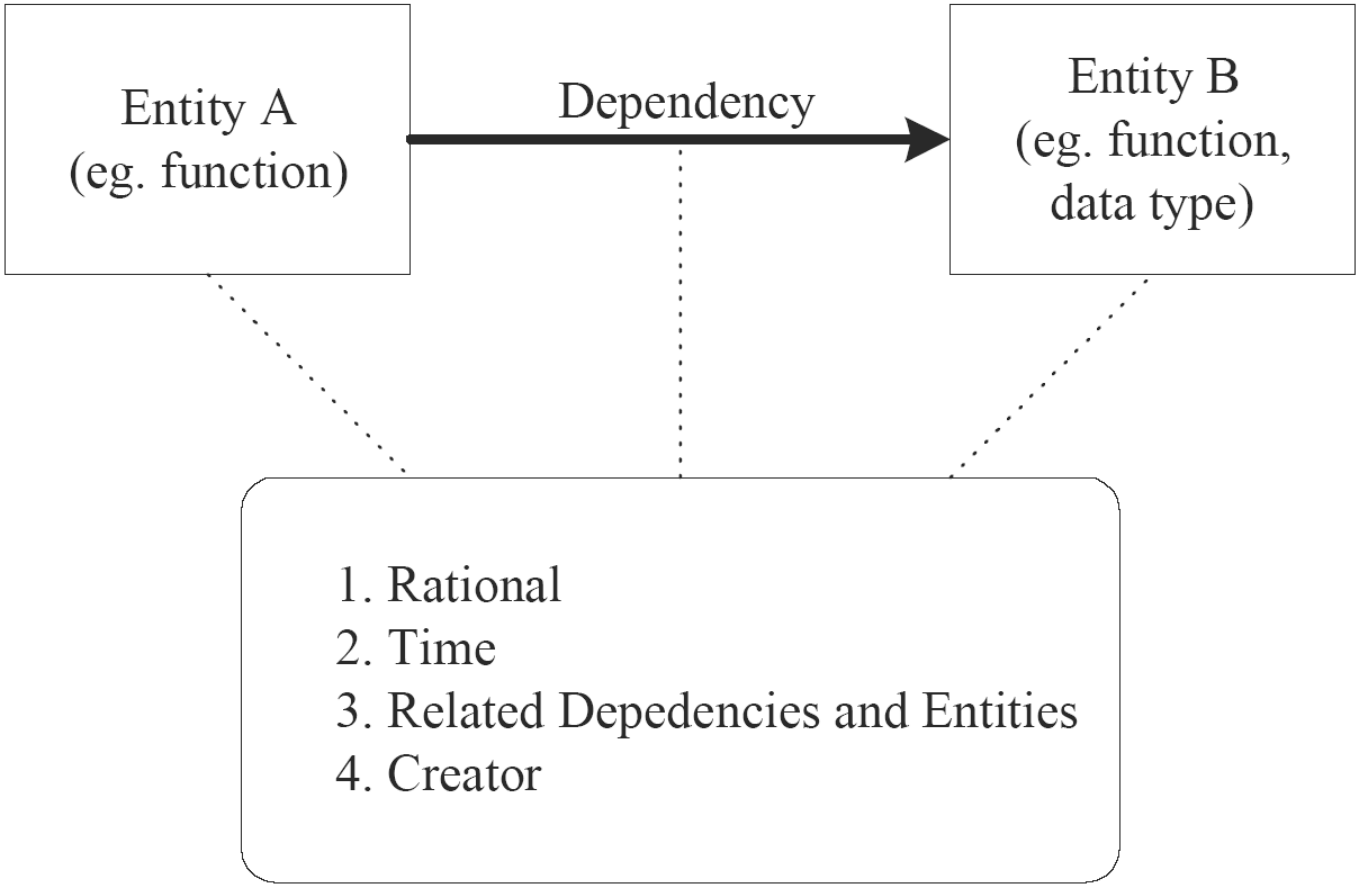
Why?

- Why was this unexpected dependency added or why was an expected dependency missing?
- A knowledge of the rationale is key in explaining the gaps

Dependency Investigation Questions (W4 Approach)

- ***Which*** low level code entity is responsible for the dependency?
  - Network (*SendData*) → Scheduler (*PrintToLog*)
- ***Who*** added/removed the dependency?
  - Junior vs. senior/experienced developer
- ***When*** was the dependency modified?
  - Late night / Just before release
- ***Why*** was the dependency added/removed?
  - The rationale!

Source Sticky Notes



- We are interested in
  - Current and past dependencies

Source *StickyNotes*

- Static dependencies give only a current static view of the system – not enough detail!
- Need to extend static dependencies, but how?

Extending Code Dependencies

- Ask developers to fill StickyNotes for each change
  - Too time consuming and cumbersome
- Use software repositories to build these notes automatically
  - Historical information may be hard to process

History as a guide

"History is a guide to navigation in perilous times. History is who we are and why we are the way we are", David C. McCullough

- Can we leverage the development history of a project in order to understand its current state?
- How can we get the development history of a project?

Challenges in studying historical code information

```
main() {
   int a;
   /*call
     help*/
   helpInfo();
}
```

V1:
Undefined func.
(Link Error)

```
helpInfo() {
   errorString!
}
  main() {
     int a;
     /*call
       help*/
     helpInfo();
  }
```

V2:
Syntax error

```
helpInfo(){
 int b;
}
  main() {
     int a;
     /*call
       help*/
     helpInfo();
  }
```

V3:
Valid code

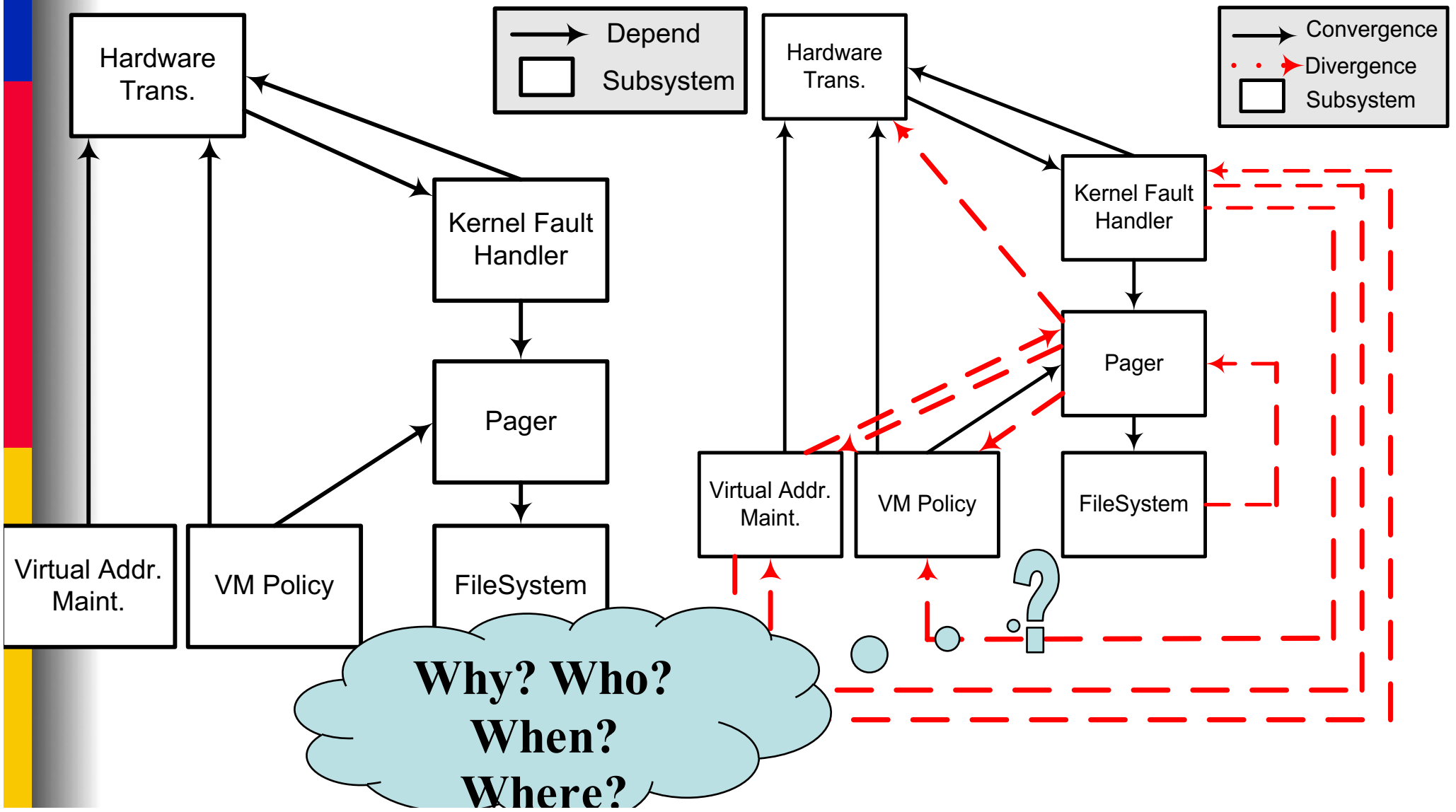
StickyNotes Recovery

- Map code changes to entities and dependencies instead of lines
- Two pass analysis of the source control repository data, to recover:
  - All entities defined throughout the lifetime of a project – ***Historical Symbol Table***
  - All dependencies between these entities and attach source control meta-data such as:
    - Name of developer performing the change
    - Text entered by developer describing the change – *the rationale*
    - Time of the change

Case Study – NetBSD

- Large long lived system with hundreds of developers
- Case study used to demonstrate usefulness of the reflexion model:
  - Reuse prior results! ☺
  - Focus on investigating *gaps* to show the strength of our approach

NetBSD Conceptual and Reflexion Model



Unexpected Dependencies

- Eight unexpected dependencies
- All except two dependencies existed since day one:
  - Virtual Address Maintenance → Pager

Which?	vm_map_entry_create (in src/sys/vm/Attic/vm_map.c) *depends on* pager_map (in /src/sys/uvm/uvm_pager.c)
Who?	cgd
When?	1993/04/09 15:54:59 Revision 1.2 of src/sys/vm/Attic/vm_map.c
Why?	from sean eric fagan: it seems to keep the vm system from deadlocking the system when it runs out of swap + physical memory. prevents the system from giving the last page(s) to anything but the referenced "processes" (especially important is the pager process, which should never have to wait for a free page).

Unexpected Dependencies

- Pager → Hardware Translations

Which?	uvm_pagermapin (in src/sys/uvm/uvm_pager.c) *depends on* pmap_kenter_pgs (in src/sys/arch/arm26/arm26/Attic/pmap.c)
Who?	thorpej
When?	1999/05/24 23:30:44; Revision 1.17 of src/sys/uvm/uvm_pager.c
Why?	Don't use pmap_kenter_pgs() for entering pager_map mappings. The pages are still owned by the object which is paging, and so the test for a kernel object in uvm_unmap_remove() will cause pmap_remove() to be used insteadof pmap_kremove(). This was a MAJOR source of pmap_remove() vs pmap_kremove() inconsistency (which caused the busted kernel pmap statistics, and a cause of much locking hair on MP systems).

Unexpected Dependencies which existed in the past

- Two unexpected dependcies that were removed in the past:
  - Hardware Translation → VM Policy
  - File System → Virtual Address Maintenance

Which?	mfs_strategy (in.src/sys/ufs/mfs/mfs_vnops.c) *depends on* vm_map (in src/sys/vm/Attic/vm_map.h)
Who?	thorpej
When?	2000/05/19 20:42:21; Revision 1.23 of src/sys/ufs/mfs/mfs_vnops.c
Why?	Back out previous change; there is something Seriously Wrong.

StickyNotes Usage Patterns

- ***First note*** to understand the reason for unexpected dependencies
- ***Last note*** to study missing dependencies
- ***All notes*** when first and last notes do not have enough information to assist in understanding

Limitations

- Quality of comments and text entered by developers in the past
- In many open source projects, CVS comments are used for:
  - Communicating new features
  - Narrating the progress of a project

Conclusions

- Development history can help understand the current structure of a software system
- Traditional dependency graphs and program understanding models usually do not use historical information
- Proposed *StickyNotes* and presented a case study to show the strength of the approach