Structural Models for Large Software Systems

Excerpts from Research Presentation

by

Murat Kahraman Gungor

Ph.D. Candidate

Advisor: James W Fawcett, Ph.D.

Introduction

- Software is expensive.
- Software projects typically consist of many parts.
- Interdependency between parts of a project is necessary.
- However, excessive dependency reduces:
  - Testability
  - Maintainability
  - Reusability
  - Understandability
- Monitoring current state of a project is critically important.

Goals of this Research

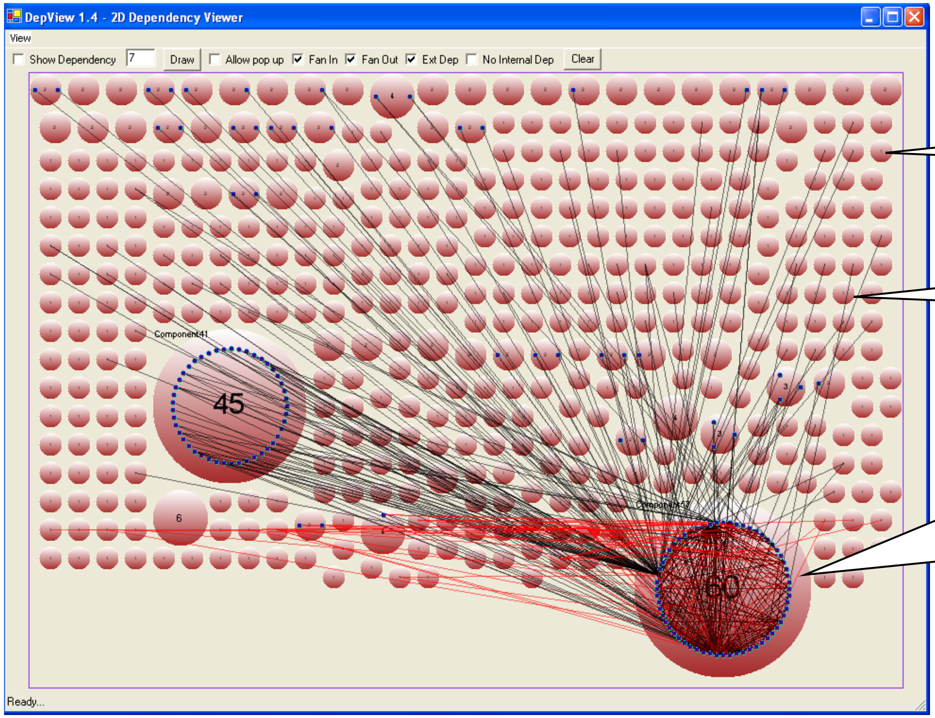
- Understand how to detect problems in large software development projects.
- Generate algorithms and methods to diagnose specific structural flaws.
- Provide tools needed to support:
  - Analysis
  - Project monitoring
- Explore possible corrective procedures and simulate their application, monitoring improvements in observed defects

A Real System

- Open Source Mozilla Project
  - Browser
    - Grew out of Netscape Navigator
  - We studied Mozilla, Windows build, version 1.4.1
    - This code base was abandoned.
    - Great opportunity to investigate why code fails.
    - After surviving serious problems, some of this code migrated into Firefox, an obviously successful implementation.
  - Windows build consists of **6193** files – for a browser!

Dependencies in GKGFX

Mozilla Rendering Library – One of many libraries



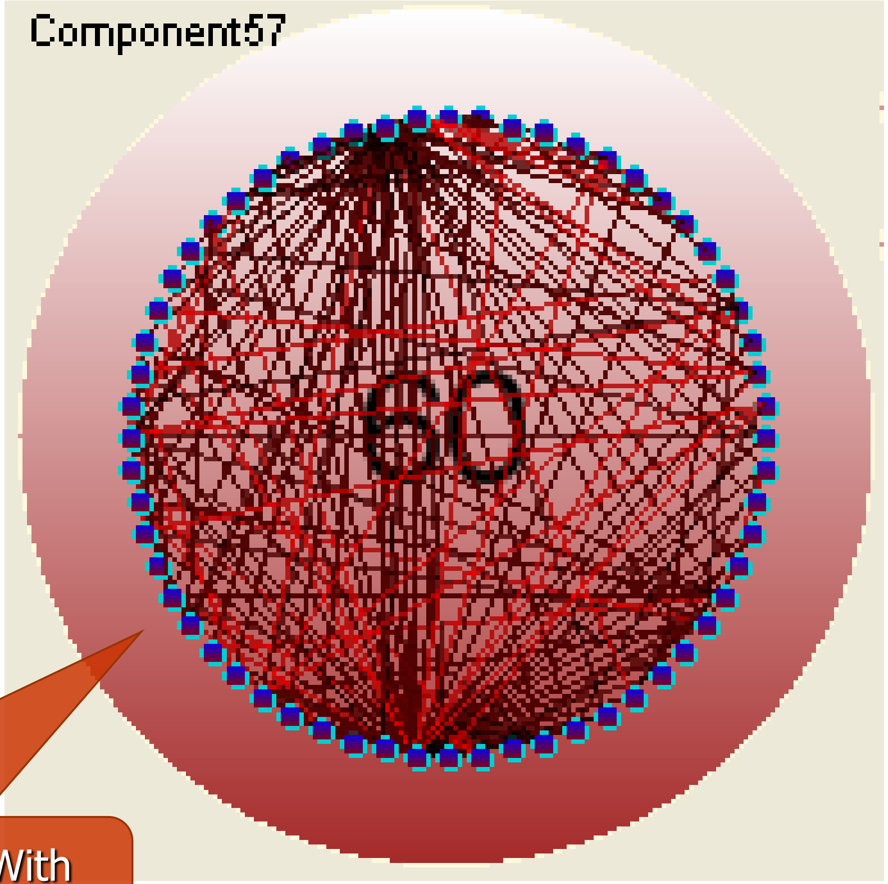
Smallest disks are single files

Lines indicate dependency

Large disks are mutually dependent files, strong components of the dependency graph

GKGFX Component Internals

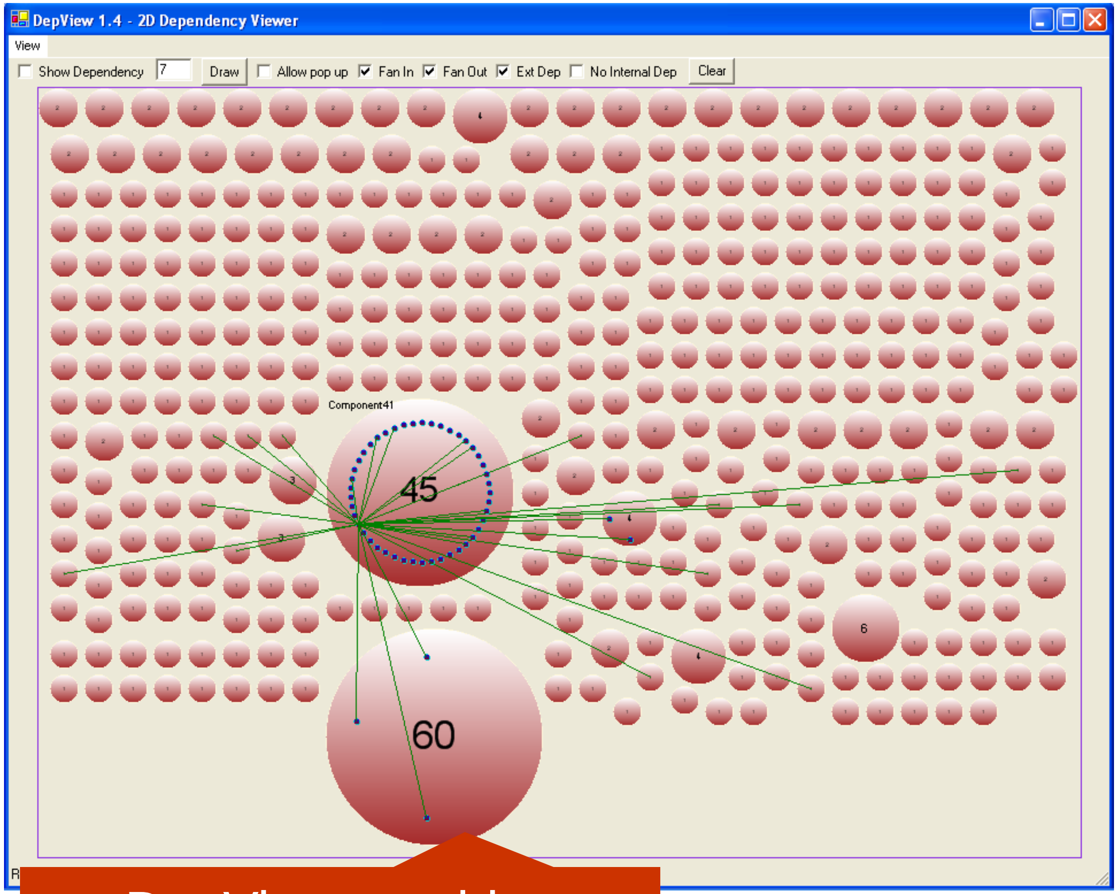
- Here are the internal dependencies for largest strong component.
- We show, in the dissertation document using Product Risk Model, that high density of dependencies within a strong component is a serious design flaw.



What's the problem? We don't know. With DepAnal and DepView, we find out.

This is Mozilla, Version 1.4.1, Windows Build

Plot for GKGFX Library shows some very large mutual dependencies



DepView provides precise definition of each strong component.

- DepView shows that the GKGFX Library does indeed have significant structural problems, as predicted by the preceding views.
- Note that these problems, made visible by our tools, are normally **invisible**!

Problem Definition

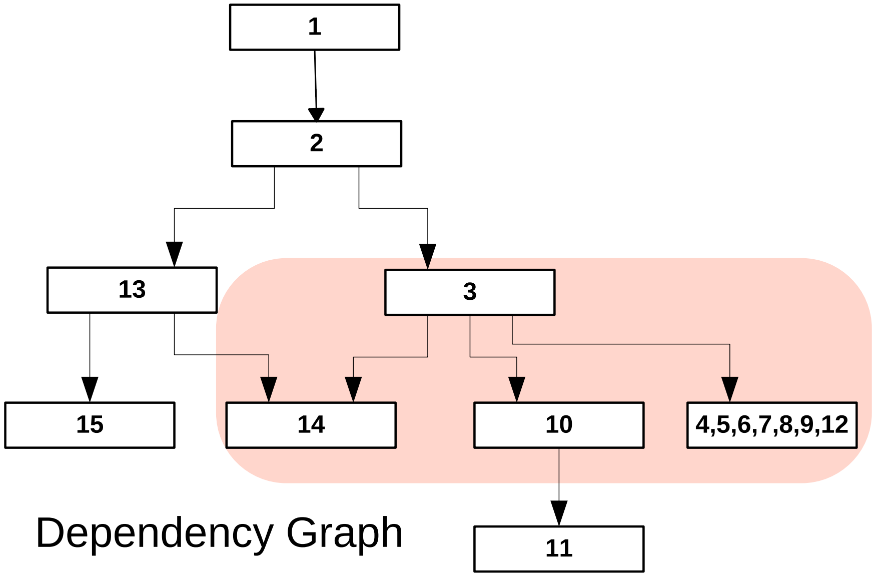
- Dependencies between software files are essential.
- However, dependencies complicate process of making changes.
- Excessive dependency degrades flexibility.
- A change may cause new changes in dependent files.

Exploring Dependency Structure

- The next few slides explain our representation of dependency
  - We discuss several kinds of dependencies that will be important later in the presentation.

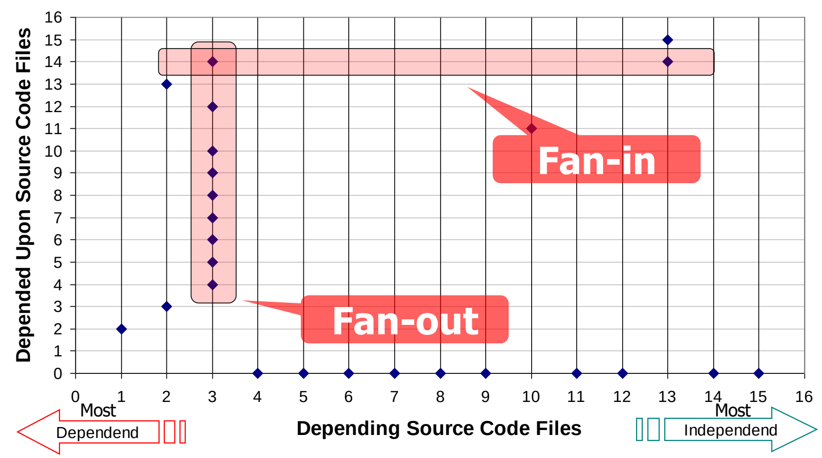
File Dependency Relationships

How to Read



Dependency Graph

After topological sort



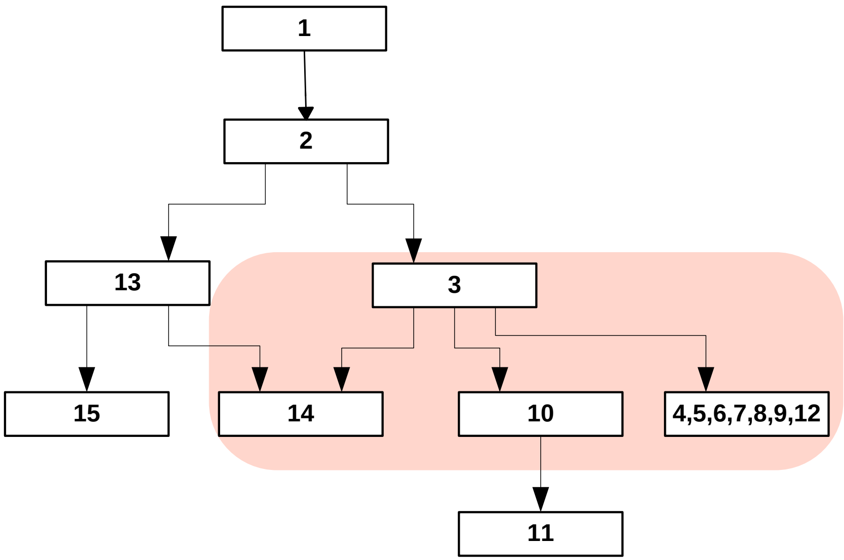
- Above shows file dependencies.
- Upper right shows another view:
  - All dots on the vertical line rooted at 3 are files that file 3 depends on. We call this Fan-Out.
  - Both dots on horizontal line rooted at 14 are files that depend on 14. We call this Fan-In.

Numbered files to the right depend only on files above them, but do not necessarily depend on every file above.

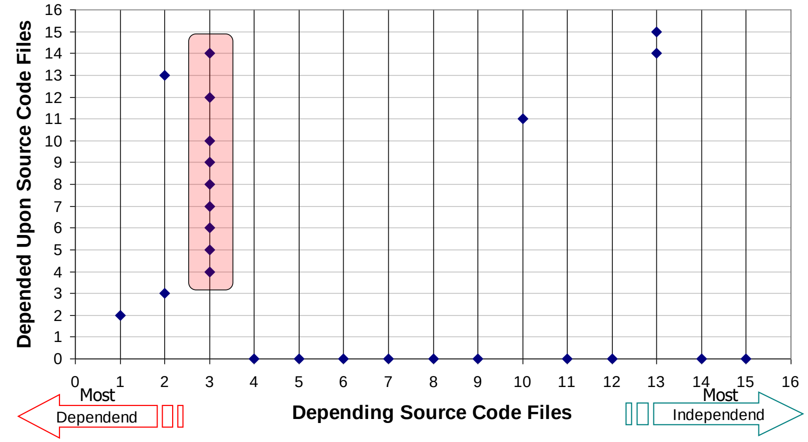
15
14
13
4
11
10
7
5
6
8
9
12
3
2
1

Top. Sorted Files

Problem: Large Fan-out



Dependency Graph - Large Fan-out

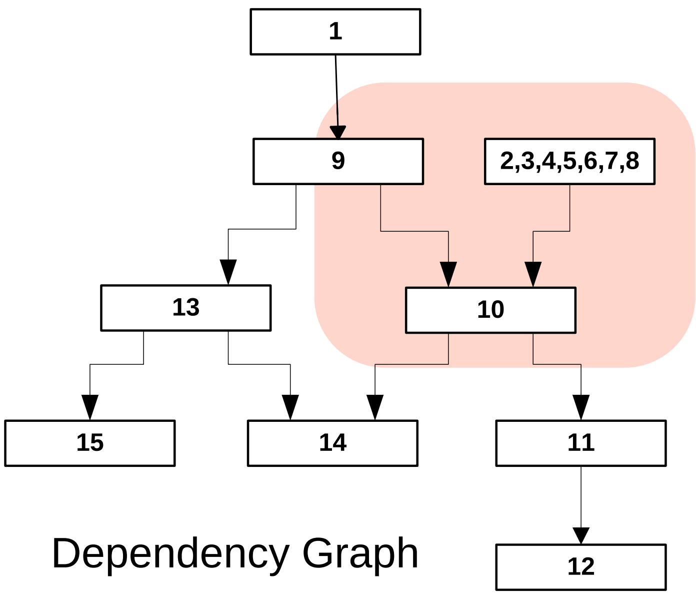


15
14
13
4
11
10
7
5
6
8
9
12
3
2
1

Top. Sorted Files

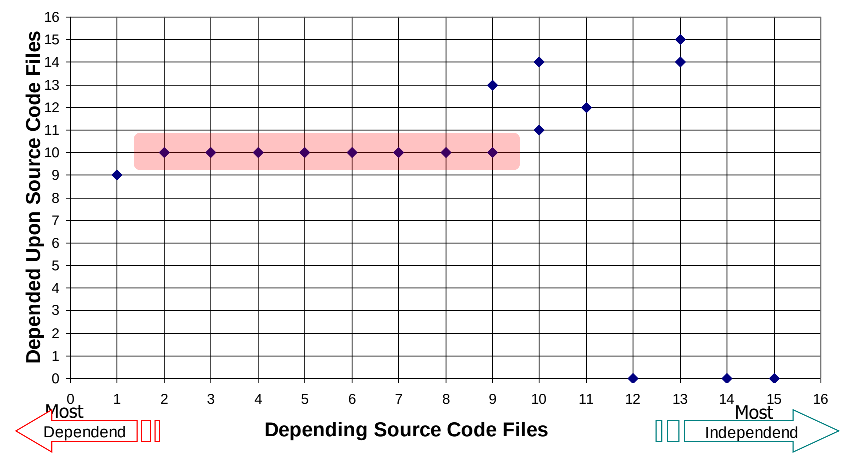
- Depending on scores of other files (large fan-out) may indicate a lack of cohesion – the file is **taking responsibilities for too many**, perhaps only loosely related, tasks and needs the services of many other files to manage that.
- Numbered files at the left depend only on files above them, but do not necessarily depend on every file above.

Problem: Large Fan-in



Dependency Graph

After topological sort



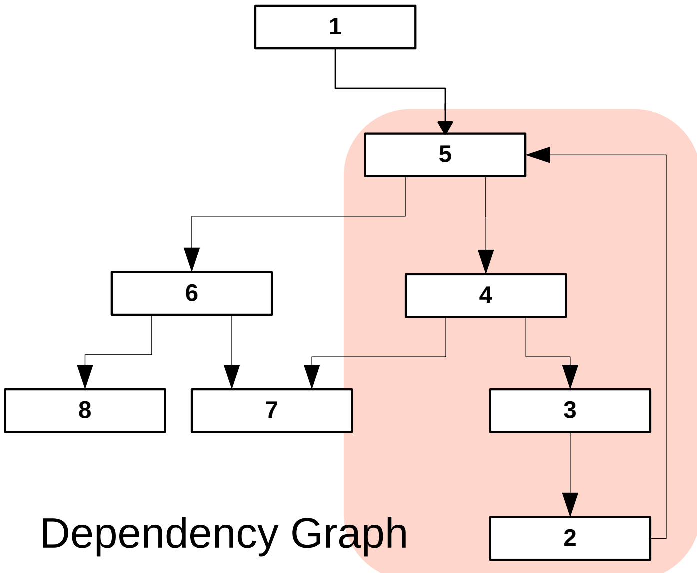
Top. Sorted Files

15
14
13
12
11
10
9
2
3
4
5
6
7
8
1

- High Fan-in is not inherently bad. It implies significant reuse which is good. However **poor quality** of the widely used file will be a problem.
- High fan-in coupled with low quality creates a high probability for **consequential change**. By consequential change we mean a change induced in a depending file due to a change in the depended upon file

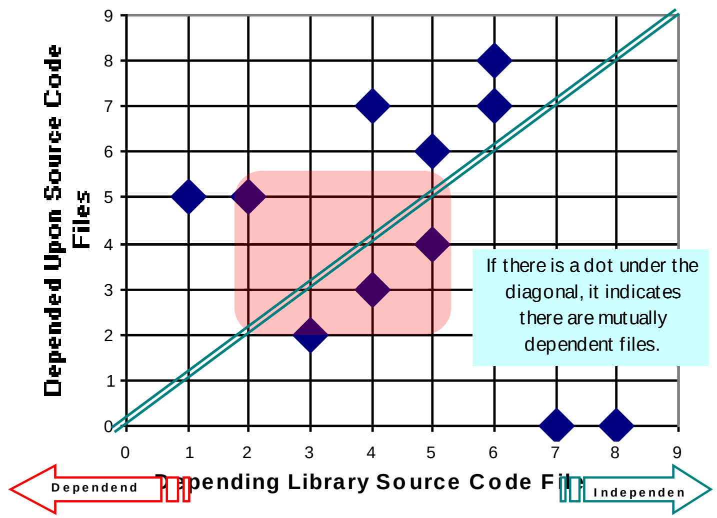
Problem: Large Strong Components

Strong component is a set of mutual dependencies



Dependency Graph

After topologically sorting, strong components are expanded



Top. Sorted Files

Files 2, 3, 4, and 5 cannot be ordered. The order given is as good as possible.

8
7
6
3
2
5
4
1

Strong Comp. Size	# of SC with this size
1	4
4	1

Ideal testing process:

- Test those files with no dependencies, then test all files depending only on files already tested.
- For testing, a strong component must be treated as a unit. The larger a strong component becomes, the more **difficult** it is to adequately **test**.
- Change management becomes tougher, due to **consequential changes** to fix latent errors or performance problems

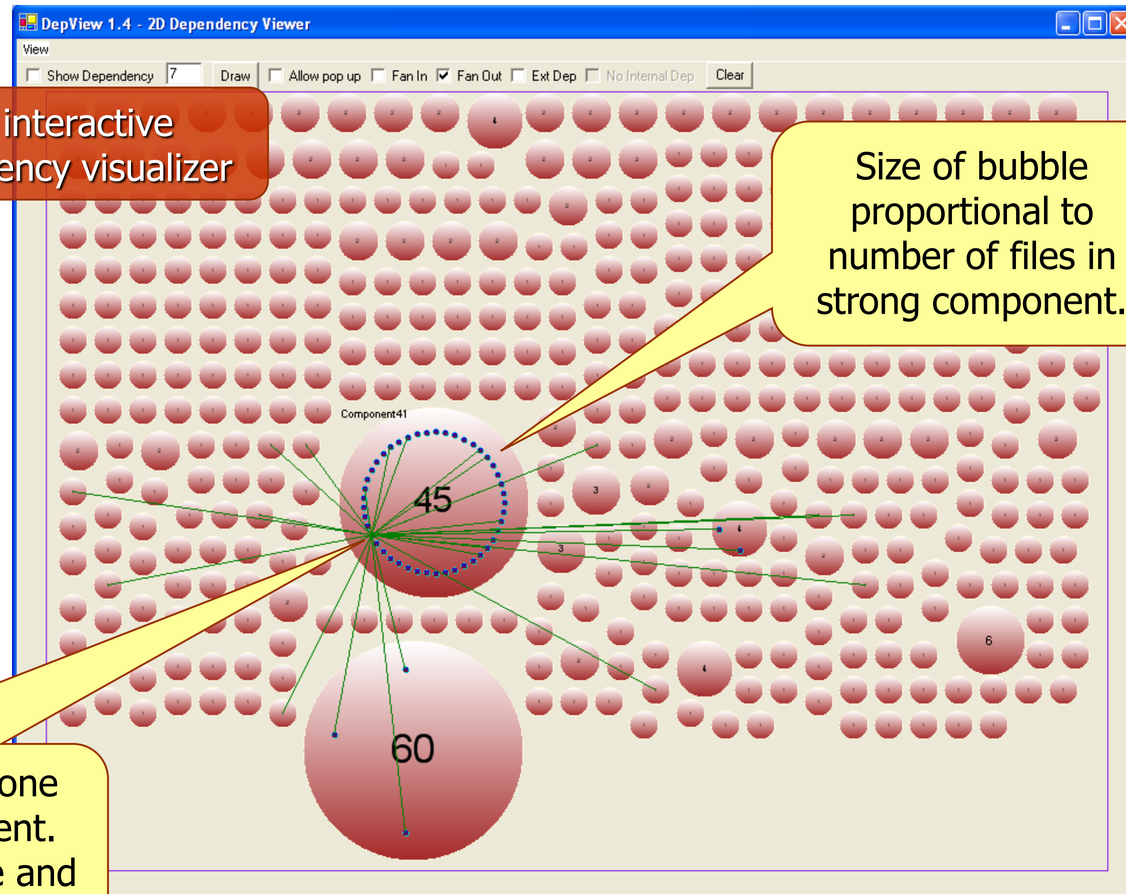
This is Mozilla's GKGFX Rendering Library

Plot shows some very large mutual dependencies

- This view is generated by our tools:
  - DepAnal
  - DepView
- This library has 598 files.
- It shows a file in a second largest strong component that depends on many other files.

Our dependency analyzer tool

Our interactive dependency visualizer

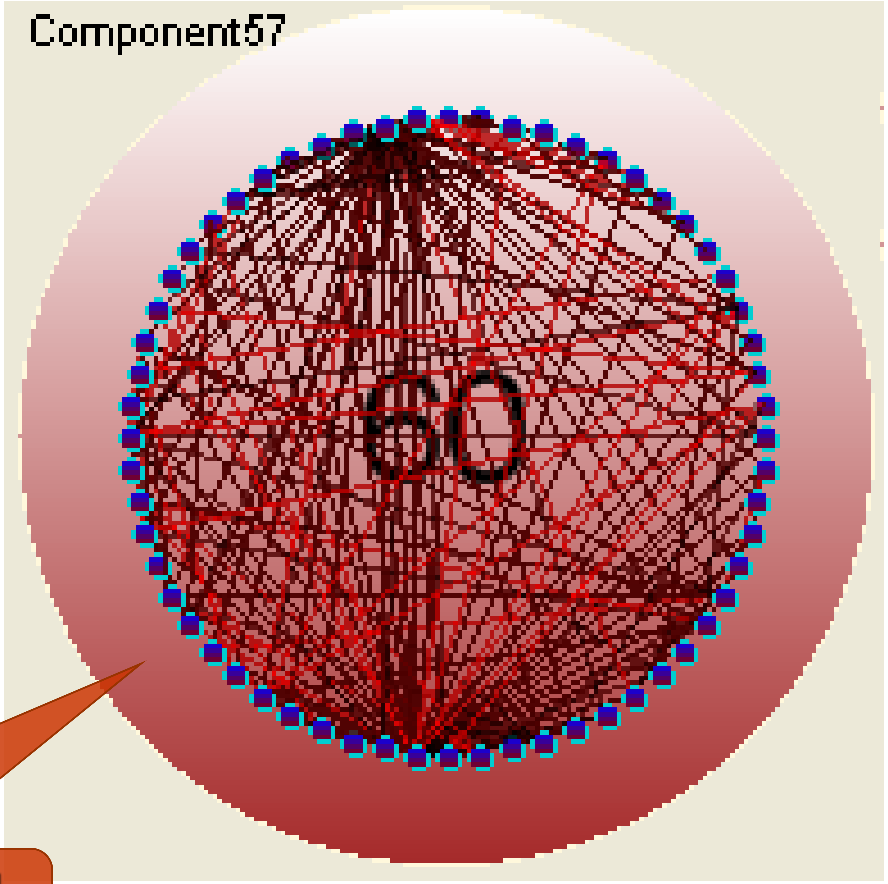


Size of bubble proportional to number of files in strong component.

Green lines show Fan-Out of one file in a large strong component. Note dependencies both inside and outside component.

GKGFX Component Internals

- Here are the internal dependencies for largest strong component.
- We show, in the dissertation document using Product Risk Model, that high density of dependencies within a strong component is a serious design flaw.



What's the problem? Without DepAnal and DepView, we don't know.

Visibility

- The dependencies shown on the previous slide are, without our tools, **invisible**.
- Developers know only a small part of the dependency structure based on their own reading of the code. The rest they **may** find by observing breakage when they change something.
- Note that Mozilla, 1.4.1 is composed of 6193 files! Impossible to understand that dependency structure without effective tools.

Is Complex Dependency Really a Problem?

- Mozilla was targeted for Apple OSX.10 but Apple switched to KHTML:
  - “Apple snub stings Mozilla” – CNET News.com
    - “Bourdon said Safari engineers looked at size, speed and compatibility in choosing KHTML.”
    - "Translated through a de-weaselizer, (Melton's e-mail) says: 'Even though some of us used to work on Mozilla, we have to admit that the Mozilla code is a gigantic, bloated mess, not to mention slow, and with an internal API so flamboyantly baroque that frankly we can't even comprehend where to begin,'" Zawinski wrote.
    - http://news.com.com/2163e+snub+stings+Mozilla/2100-1023_3-980492.html

Our Approach

- Having seen the previous problems, here is what we are going to do.

Scope of Study

- We are **not** analyzing syntactic correctness of code.
- We are **not** analyzing logical correctness of code.
- We **are** analyzing project code structure.
- Our methods and tools are **applicable** to C-based procedural and object oriented languages such as C, C++, C#, Java.
  - DepAnal and DepView support both C and C++

Contributions

- Developed Source File Ranking Models
  - Risk Model,
  - Reusability Index.
- Developed Analysis Methods
  - Dependency Analyzer (DepAnal): C/C++ static source code dependency analyzer tool. Able to analyze thousands of files in reasonable time (Mozilla: 6193 files in approximately 4 hours – dependency and graph relationships).
  - Dependency Viewer (DepView) – Interactive visualization of dependencies among files and components. Provides new views of complex information.
- Designed and conducted an experiment to investigate the impact of change in one file on other files (results shown later).
- Investigated corrective procedures and simulated their application, monitored improvements in observed defects.

Dependency Model

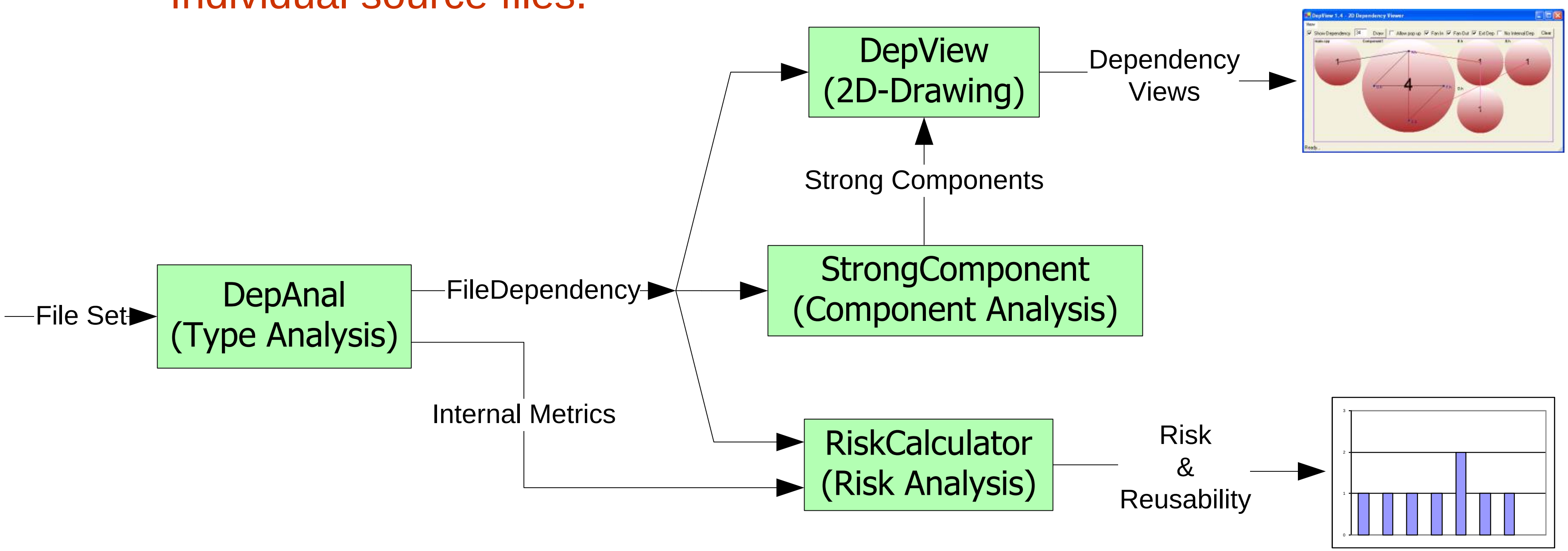
summary

- Focus is dependencies between files.
  - Files are unit of testing and configuration management
- Based on types, global functions and variables.
  - Dependency Model - file A depends on file B if:
    - A creates and/or uses an instance of a type declared or defined in B
    - A is derived from a type declared or defined in B
    - A is using the value of a global variable declared and/or defined in B
    - A defines a non-constant global variable modified by B
    - A uses a global function declared or defined in B
    - A declares a type or global function defined in B
    - A defines a type or global function declared in B
    - A uses a template parameter declared in B
- Outputs are presented as direct dependencies.
  - We do not show transitive closure for ease of interpretation – otherwise, too dense.
- Risk model accounts for transitive relationships, in an effective way.

Data Gathering and Processing

summary

- Figure below is the data gathering and processing flow used during our analysis of software.
- We obtain data in two different granularities:
  - Strong components.
  - Individual source files.



An Analysis – Mozilla, Version 1.4.1

- The Mozilla project is a very large project developing browser tools for many different platforms.
- Win 32 Configuration
  - Number of executables: 94
  - Number of dynamic link libraries: 111
  - Number of static libraries: 303
  - Number of source files for Win32, v 1.4.1: 6193

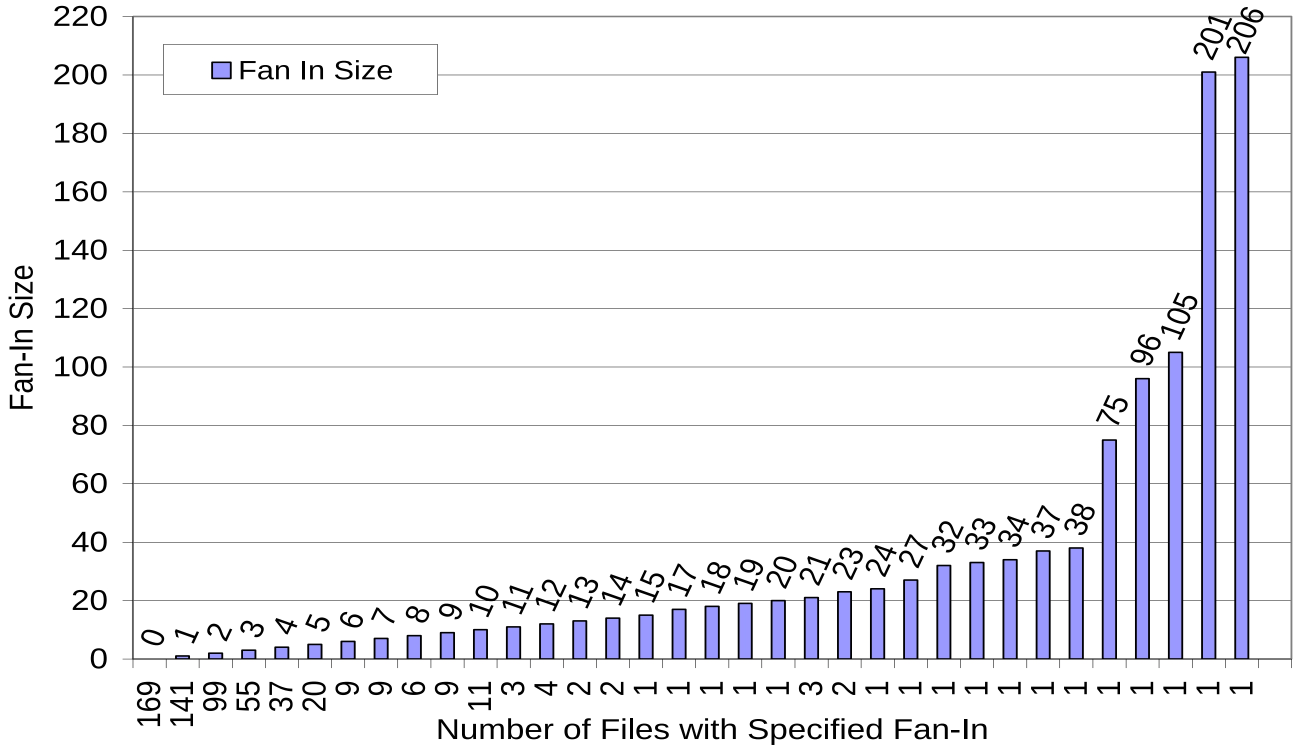
Wow!

- Analysis of entire Mozilla project took approximately 4 hours on Dell Dimension 8300 with 1 G Memory
- Can analyze individual libraries – few hundred files – in half hour.

Fan-in Density

Mozilla GKGFX Library

- This histogram shows that significant number of library source code files have high fan-in, characteristic of a widely used library.

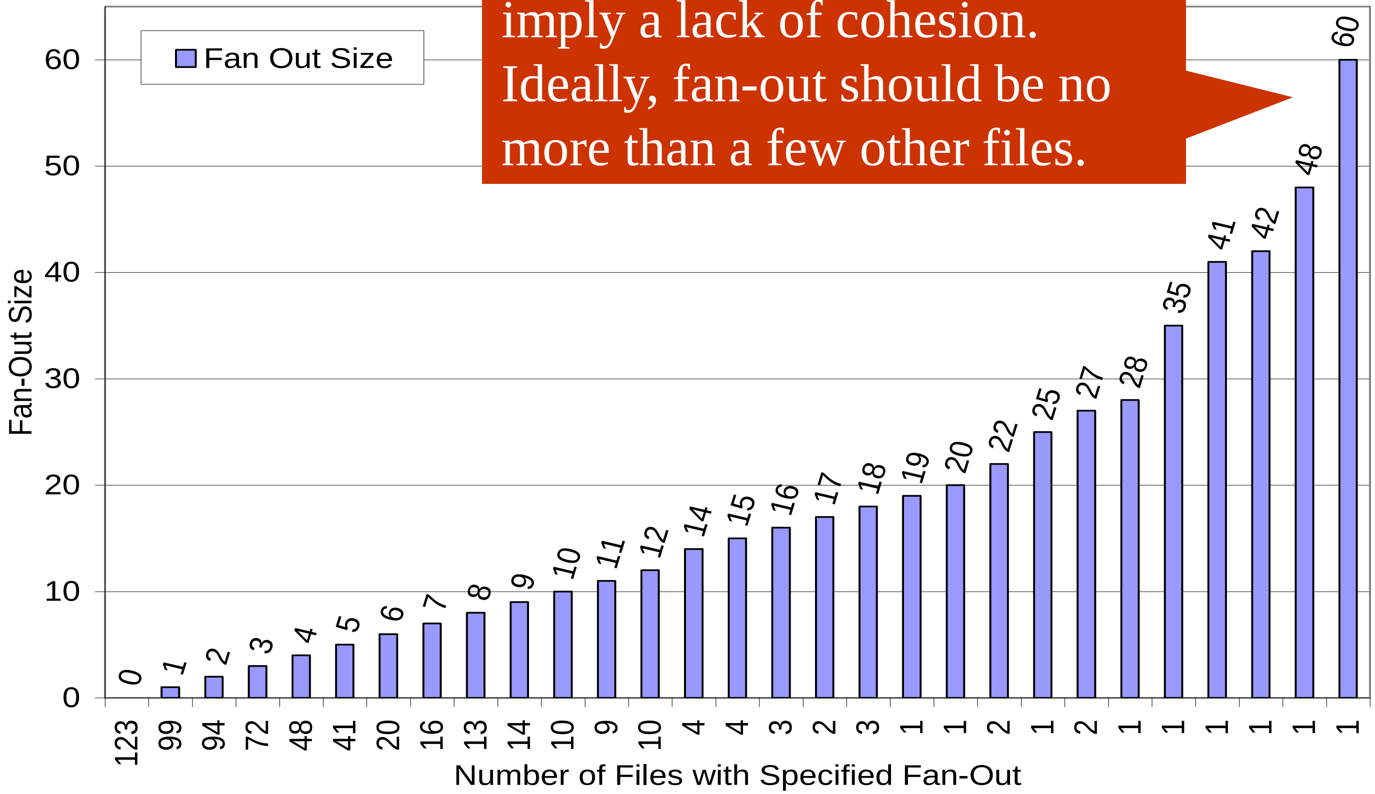


A library with this profile should be given high priority for analysis by the test team and quality analysts.

Fan-out Density Mozilla GKGFX library

- Large Fan-Out may be symptomatic of weak abstraction. We've show elsewhere that High Fan-Out is correlated with large number of changes.

Large fan-out is likely to imply a lack of cohesion. Ideally, fan-out should be no more than a few other files.



There are a significant number of files with large fan-out.

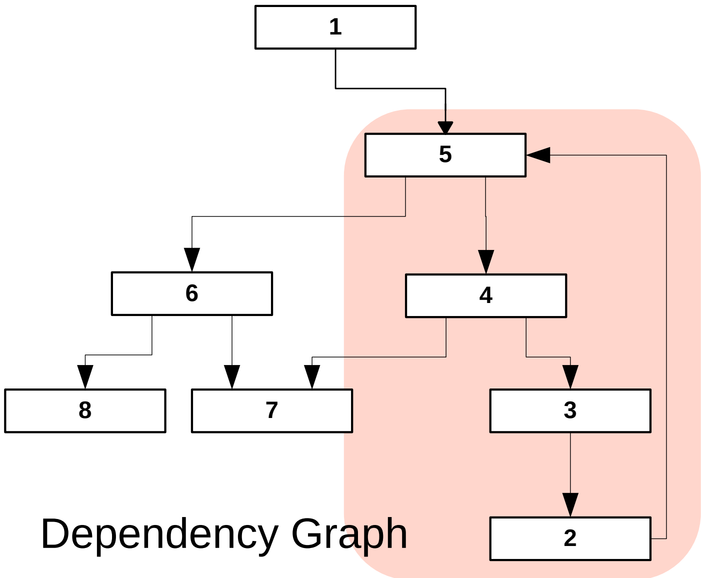
Summary for High Level Views

- High Fan-in implies:
  - Good reuse.
  - Large testing effort if we need to make a change in file with high Fan-In.
- High Fan-out implies:
  - Weak abstraction.
  - Need for redesign or refactoring of code.

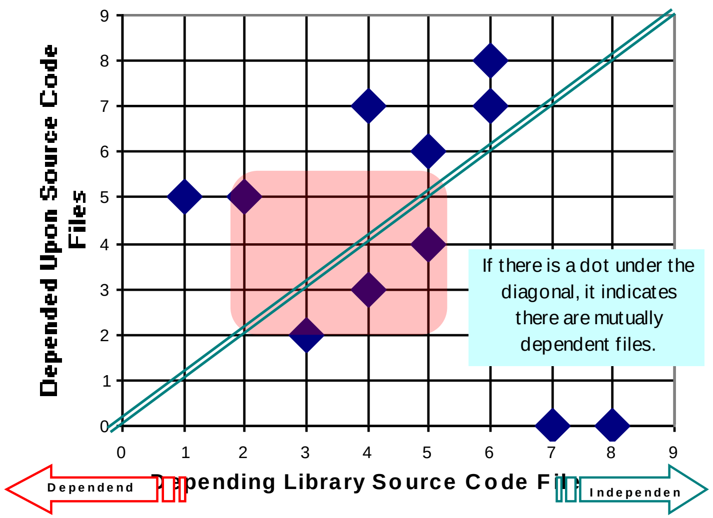
Problem: Large Strong Components

Strong component is a set of mutual dependencies

reminder

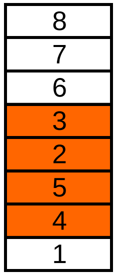


After topologically sorting, strong components are expanded



Top. Sorted Files

Files 2, 3, 4, and 5 cannot be ordered. The order given is the best we can achieve.



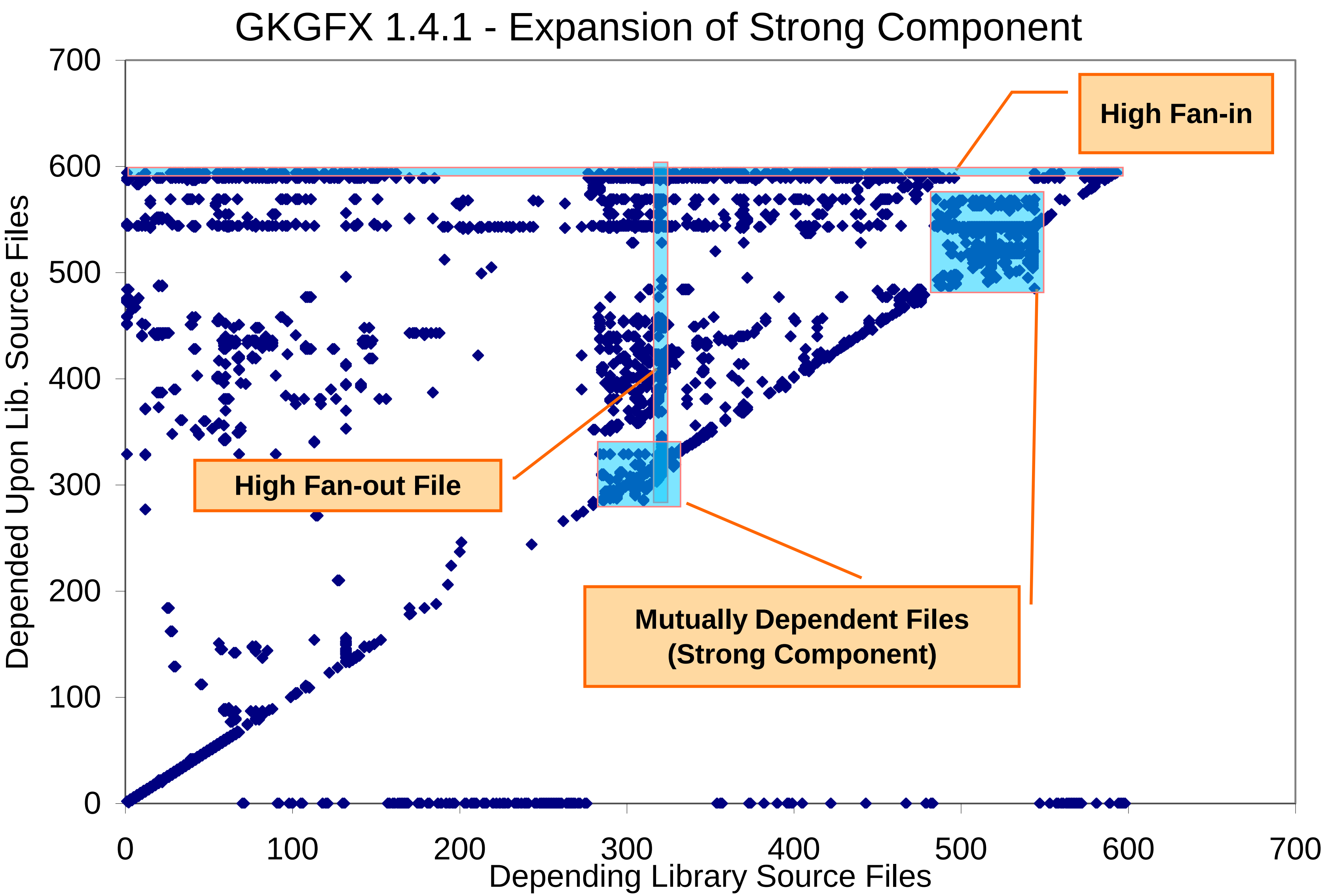
Strong Comp. Size	# of SC with this size
1	4
4	1

Ideal testing process:

- Test those files with no dependencies, then test all files depending only on files already tested.
- For testing, a strong component must be treated as a unit. The larger a strong component becomes, the more difficult it is to adequately test.
- Change management becomes tougher, due to con-sequential changes to fix latent errors or performance problems

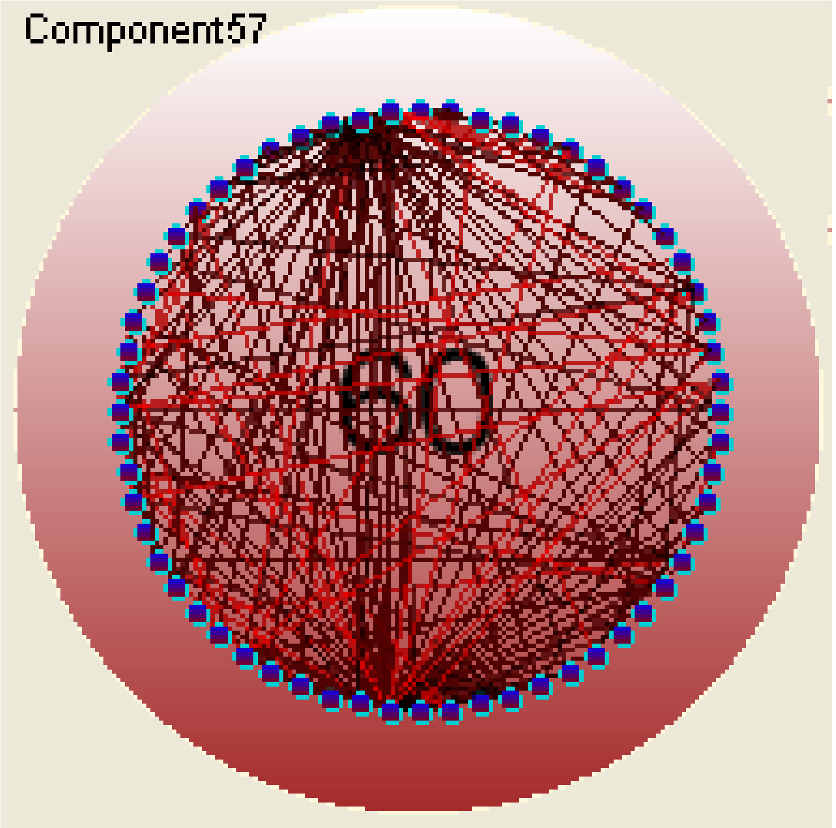
Analyzing Dependency Matrix

Topological sort gives best test order – important information!



GKGFX Component Internals

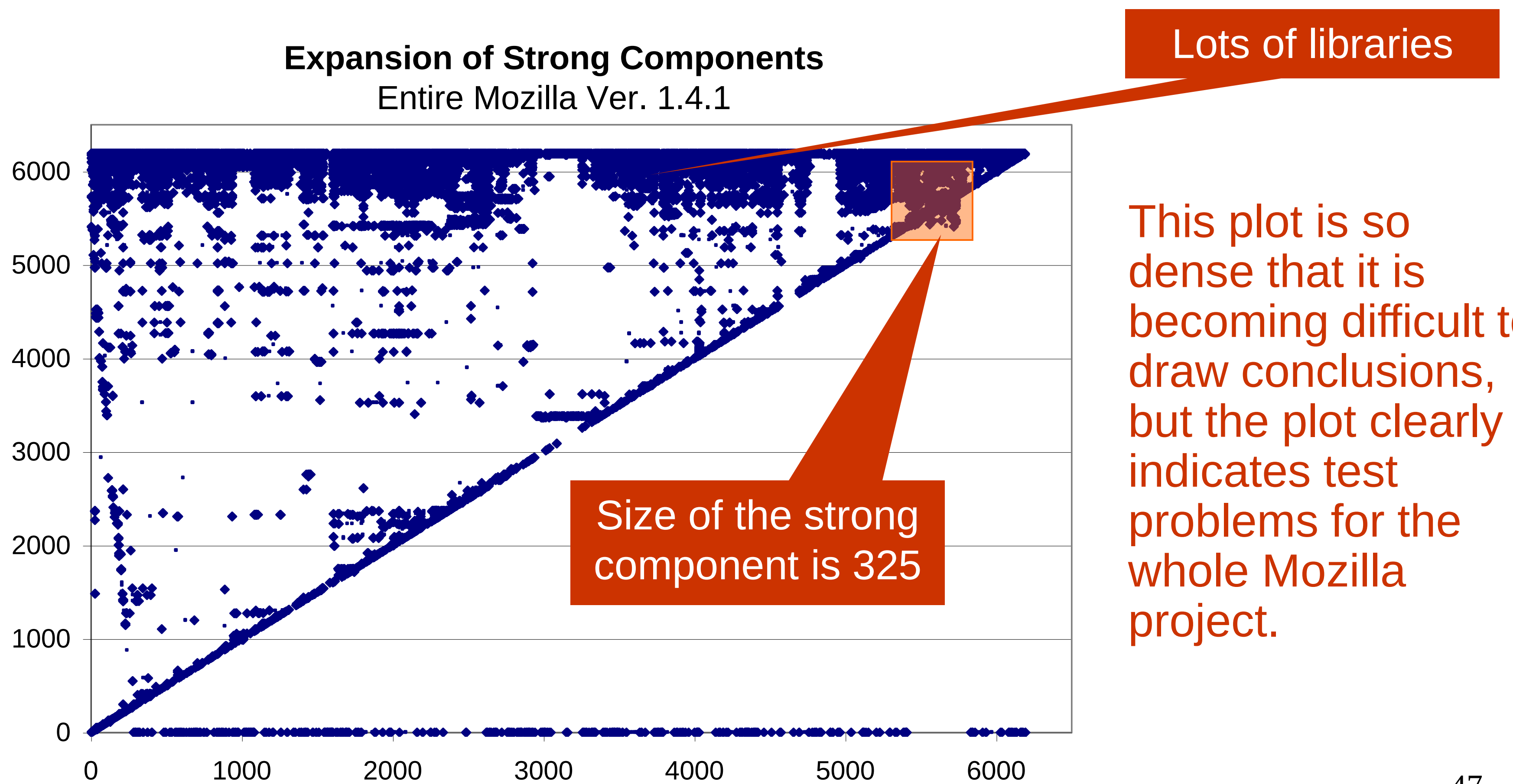
- Here are the internal dependencies for largest strong component.
- We show, in dissertation document, using Risk Model, that high density of dependencies within a strong component is a serious design flaw.



Dependency Data

For the Entire Windows-Based Mozilla Build

- The plot below is a topological sorting of the dependency graph and then expanding strong components of the entire Mozilla build for windows.



This plot is so dense that it is becoming difficult to draw conclusions, but the plot clearly indicates test problems for the whole Mozilla project.

So how do we make sense of all this?

- We’ve now seen significant problems in the Mozilla 1.4.1 structure.
- How can we find what is the cause of the problems?
- How can we find ways to improve?

Product Risk Model

- Product Risk Model is a file-rank procedure that orders the entire system’s file set by increasing risk.
  - Provides direct support for management of large developing code bases.
  - ***Indicates where attention should be focused.***
  - ***Enables developers to observe overall effect of a particular change (simulation)***
    - Removing global objects, interface insertion.

Product Risk Model

Definitions

- **Importance** of a file is based on the number of other files that directly or indirectly depended upon it.
- **Test Difficulty** is the degree of relative effort required for a file to be tested based on:
  - Number of files it is using and its interconnectedness strength,
  - Internal implementation quality

α_{mn}

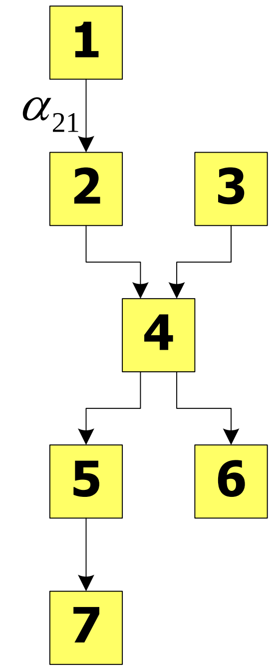
$$I_i = 1 + \sum_{AllCallers} \alpha_{ij} I_j$$

$$I \in [1, \infty)$$

$$\alpha \in [0,1]$$

$$T_n = \beta_n + \sum_{AllCalled} \alpha_{mn} T_m$$

$$T \in [1, \infty)$$



Product Risk Model

Definitions cont'd…

- Implementation Metric Factor

$$\beta_i = 1 + \frac{1}{N}\sqrt{\left(\frac{m_{1i}}{M_1}\right)^2 + \left(\frac{m_{2i}}{M_2}\right)^2 + \dots + \left(\frac{m_{Ni}}{M_N}\right)^2}$$

$$\beta_i = 1 + \frac{1}{N}\sqrt{\sum_{j \in (1,N)} \left(\frac{m_{ji}}{M_j}\right)^2}$$

M: Boundary metric value

m: Measured metric value

N: Number of metric involved

Small (m/M) is good.

- Risk of a file is the product of its importance and test difficulty.

$$R_i = I_i \times T_i$$

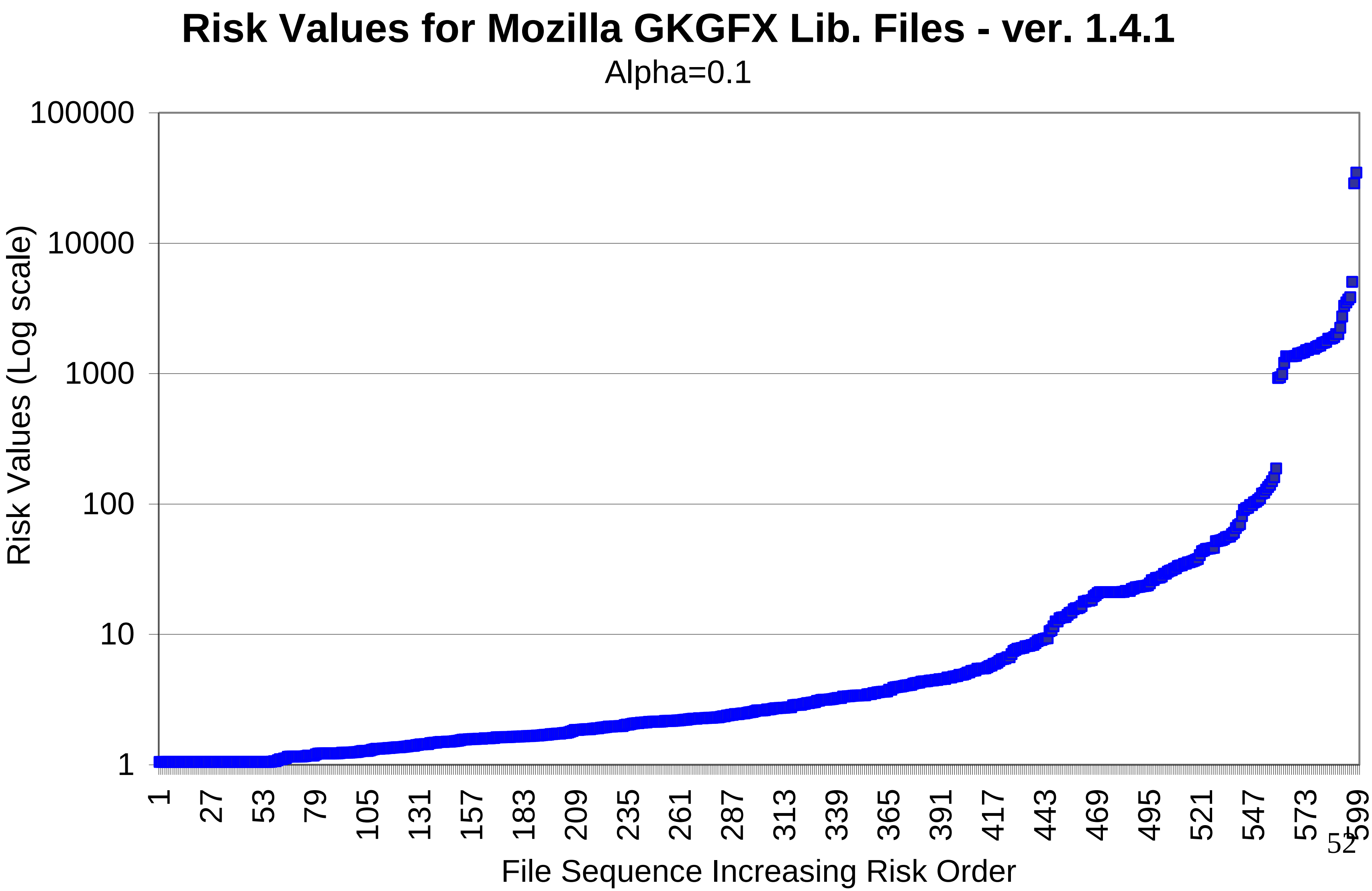
$R \in [1, \infty)$ $I \in [1, \infty)$ $T \in [1, \infty)$

Low I and low T are good

- Alpha represents the relative frequency of required consequential changes in files in the project.
- Test difficulty of a file depends not only on its internal implementation quality, but also on the quality of the files that it depends on.

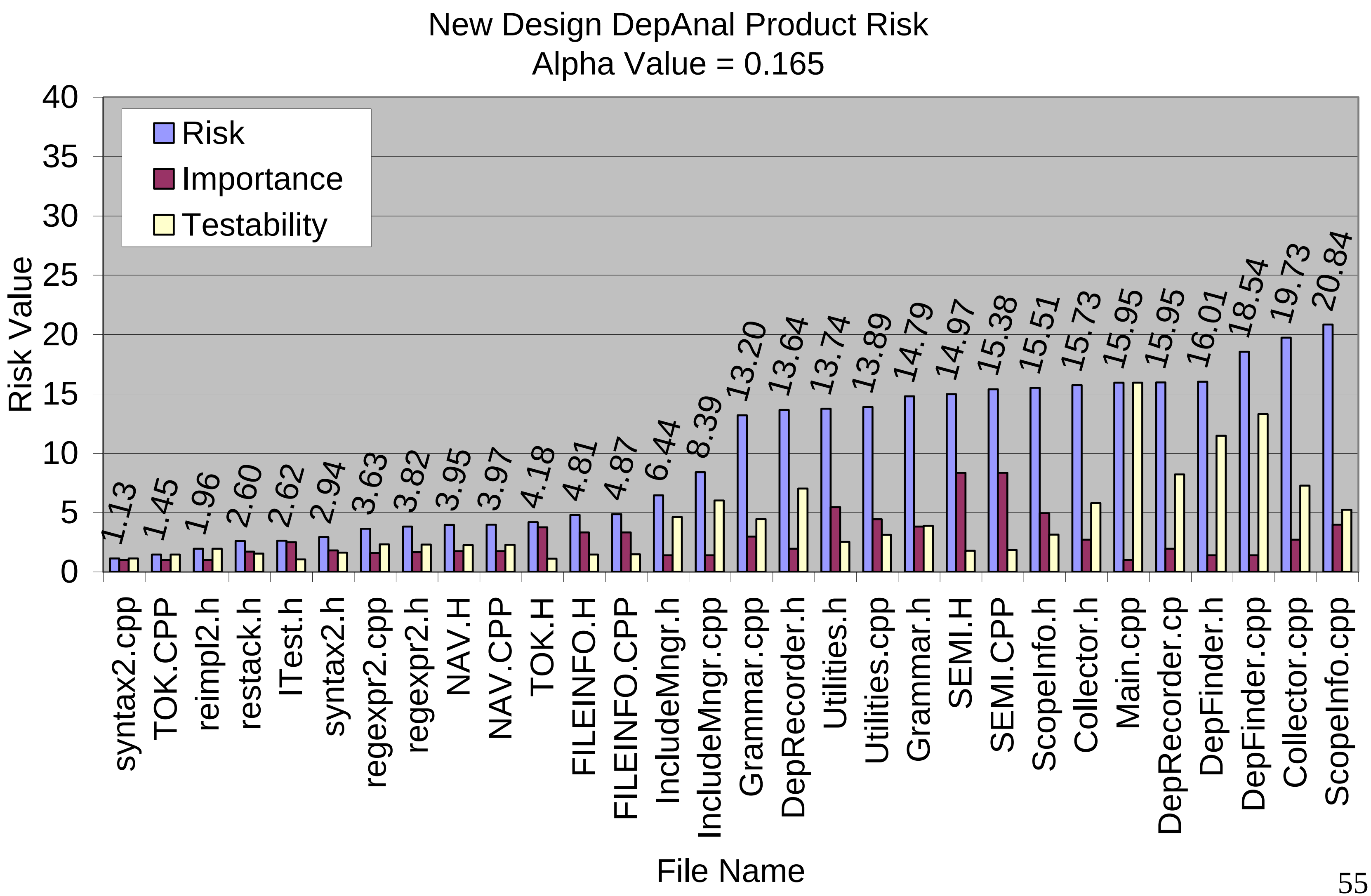
Risk Model Applied

Mozilla GKGFX Library



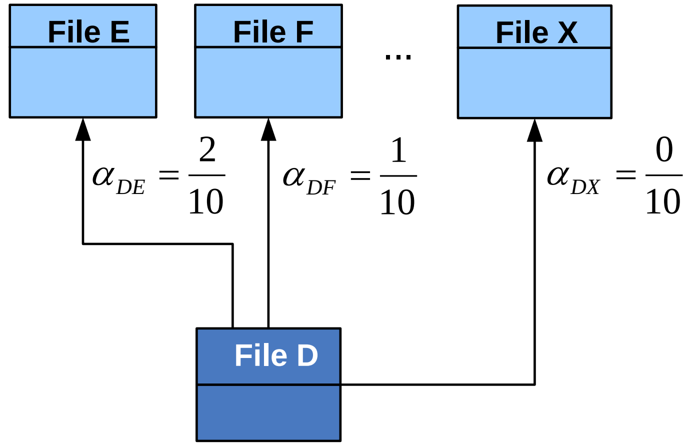
Risk Model Applied

Risk Values with File Names - New Design



Change Impact Factor (α_{ij}) Estimation

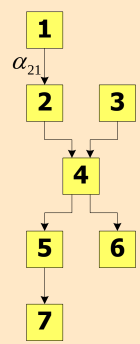
- Goals is to understand the impact of a change in a software source file to other source files
- What we did?
  - Designed an experiment,
  - Described its application,
  - Showed measured results of the change impact.
- Redesigned DepAnal
  - The analyzer's first external release has 7796 lines of new code,
  - 5580 of these are code within functions.
  - Implementation took three months, and
  - 503 changes were recorded.



$$\alpha_{DE} = \frac{2}{10} = \frac{\text{Consequential changes to E caused by changes in D}}{\text{Total changes in D}}$$

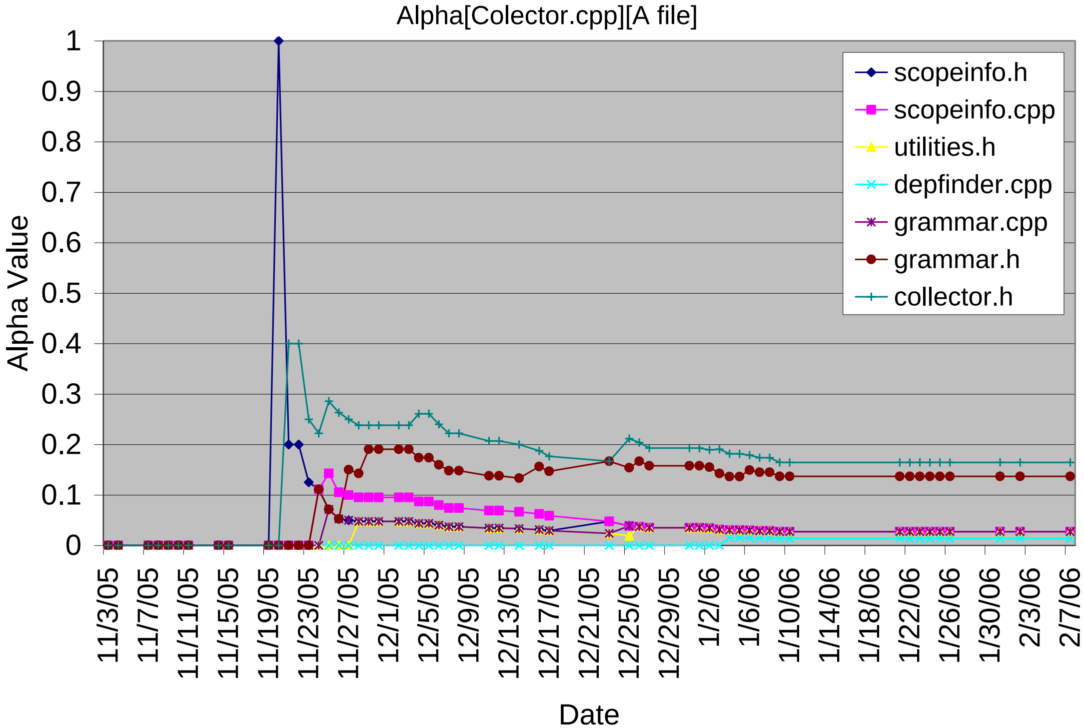
$$I_i = 1 + \sum_{AllCallers} \alpha_{ij} I_j$$

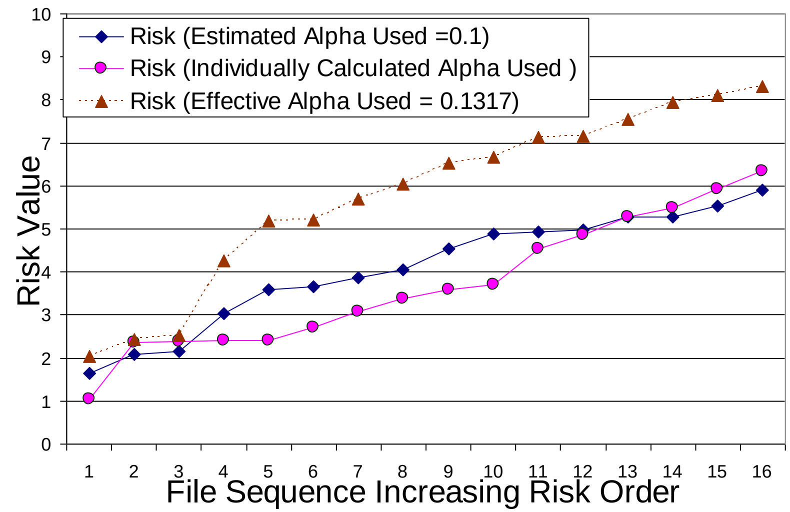
$$T_n = \beta_n + \sum_{AllCalled} \alpha_{mn} T_m$$

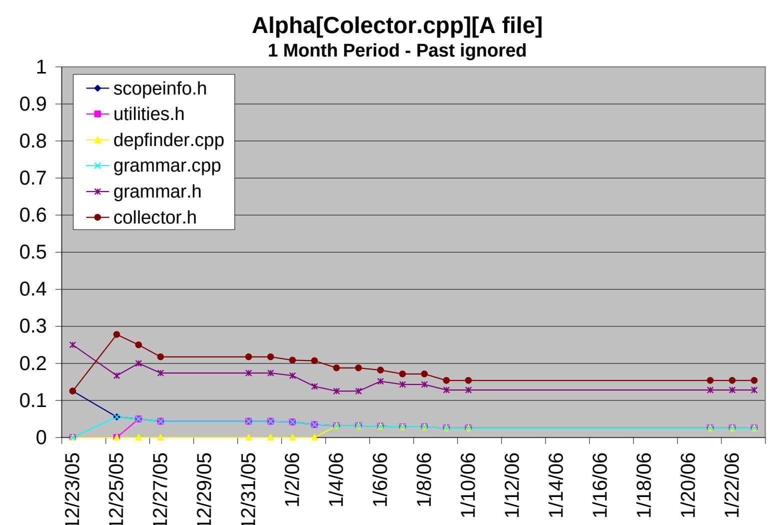


Results

Change Impact Factor







- Once reached a steady state the alpha values can be approximated by some constant factor

$$\alpha_{Collector.cpp,A} = \frac{\sum \text{Consequential change count in A due to Collector.cpp}}{\sum \text{Changes in Collector.cpp}}$$

$$\alpha_{\text{Effective}} = \frac{\sum_{Filei}^{m} \sum_{File\,j}^{n_i} \text{Consequential change}_{ij}}{\sum^{m} \text{Change}_i} \; if \text{ Consequential change}_{ij} \neq 0$$

File Reusability Ranking Model

- Reuse of previously developed software components is desirable to take advantage of work on previous projects and to avoid development effort and cost that would otherwise be required.
- This ranking model helps engineering organizations capture most important parts of a project to reuse in the future.
- Enables developers to evaluate a file for reuse without initially looking at its code. Especially for the large projects, and may be almost impossible to accomplish manually due to complex interdependencies
- There is no good way to do that without our methods and tools.

File Reusability Ranking Model Cont…

$$RI = \frac{FI}{FI + \overline{FO} + \beta}$$

$\overline{FO}$

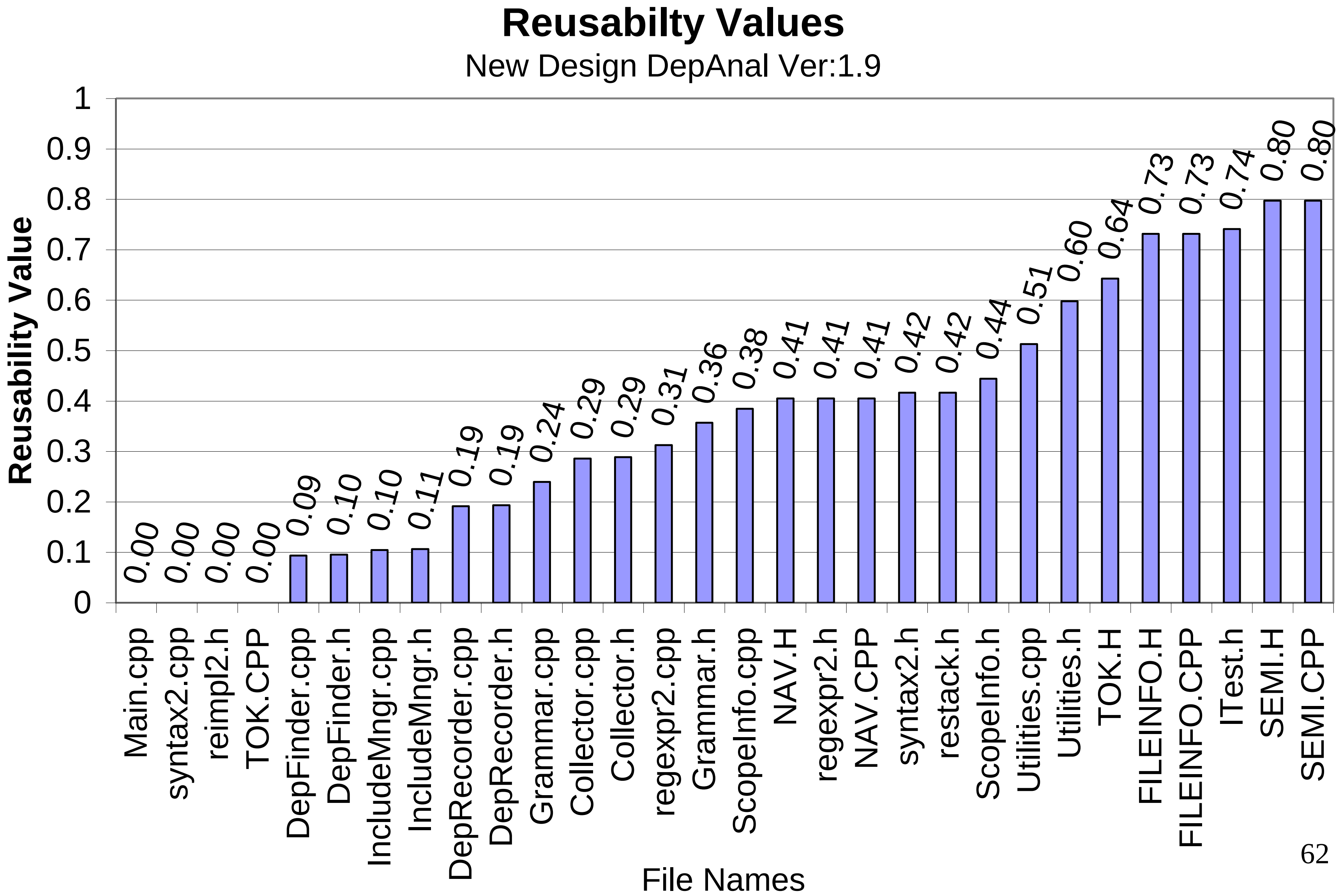
transitive closure of fan-out

$\beta \in (0, \infty)$ $RI \in [0,1)$

- High RI (close to 1) is preferred.
- If a file is called by many others in the product, e.g., has a high fan-in, then it has demonstrated its usefulness, at least within that product by this in-situ reuse.
- If, however, it has a high fan-out, then it depends on many other files, which makes it much harder to reuse.

Reusability Model Applied

DepAnal

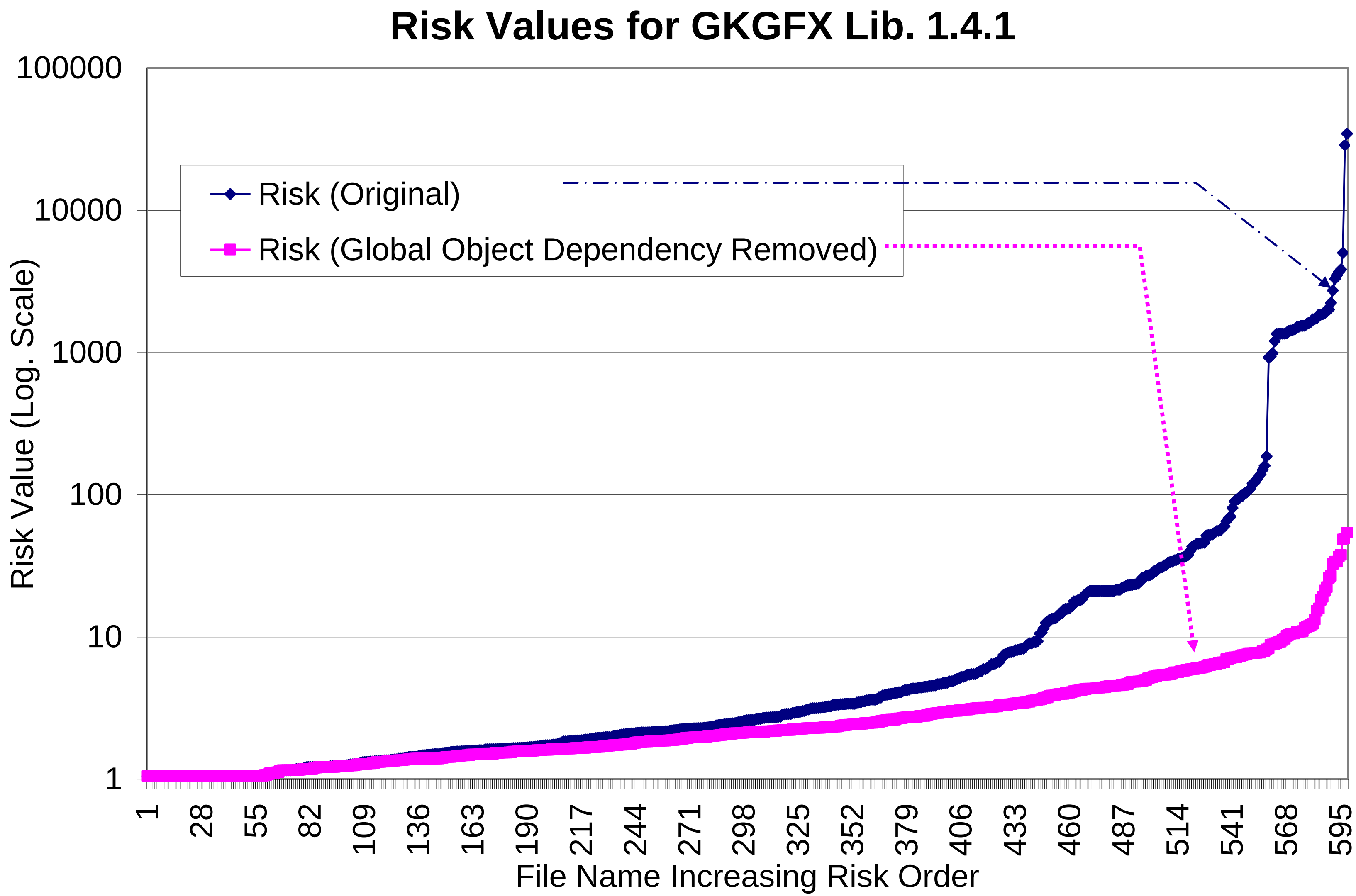


Simulating Constructive Changes

- We examine the affect of changes we may make to improve the structure of systems analyzed with the help of DepAnal and DepView
- We simulated (except for DepAnal) the effects of changes
  - Elimination of global variables and
  - Inserting interfaces between components.

Change in Risk Values

Simulation of Global Data Elimination - GKGFX



Conclusions to this Point

- The models and tools we've developed for this research have the power to find and display structural problems in large software systems.
- Our work shows that specific constructive changes can significantly improve system structure and reduce risk.

Contributions

- **Developed Risk model** which **pinpoints problem files** and supports comparisons before and after fixes.
- We **introduced a reusability model** that indexes software components according to their **potential for reuse**.
- We designed and **conducted an experiment** to investigate the **impact of change** in one file on other files, in terms of consequential changes they require.
- We designed and **developed tools** implementing these algorithms and methods that are **capable of analyzing very large sets** of files (6193 files analyzed in 4 hours)
  - DepAnal/DepView is our experimental apparatus needed to provide new results.
- Demonstrated **specific means to improve structural problems**, using risk model and DepAnal/DepView.