University of Sheffield

Metrics and Design Quality

Software Reengineering
(COM3523 / COM6523)

The University of Sheffield

Design that is easy to understand.

Recent field study of seven large industrial projects:

- Involving 78 professional developers, spanning 3,148 hours of dev-time
- **Developers spend ~60% of their time on "program comprehension".**
- **A further ~25% of their time is spent on navigation.**

Good design is essential.

- Reduces "lost time" on understanding what is already there.
- Reduces risk of misunderstandings, and introduction of bugs.

Key to good design: Modularity

- How well are domain concepts "packaged" into classes?

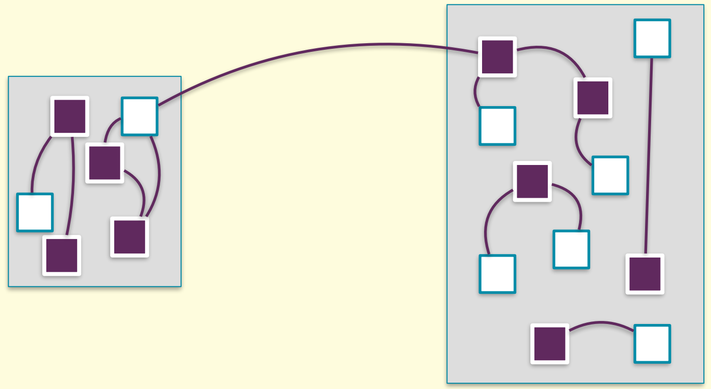
Measuring program comprehension: A large-scale field study with professionals, Xia, Bao, Lo, Xing and Hassan, TSE 2018

Coupling and Cohesion: Good Modularity

Coupling

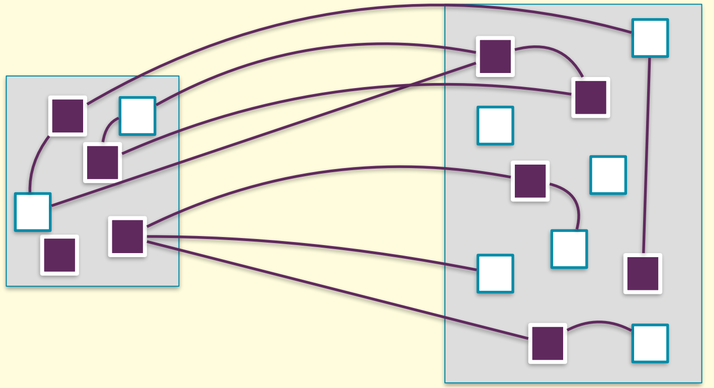
"Tightly coupled" if there are lots of calls or data-accesses across file / module boundaries.

Loose coupling encouraged by **encapsulation** - good use of **public interfaces**.



Cohesion

A single file or module is "cohesive" if the code within it is highly interdependent.



Goal: Minimise coupling, maximise cohesion.

Class / Module Size

Why is size relevant?

- Individual classes should have a single responsibility.
  - This guideline is known as the **Single Responsibility Principle**.
- Should minimise duplicate code.
  - This guideline is known as **Do Not Repeat Yourself (DRY)**.

Big classes (lots of methods, high LOC) suggest that these principles have been violated.

- Tend to lack cohesion, and incur high coupling.
- Hard for developers to fully understand - difficult to maintain.
- Defect prone.
- “God classes”.

SOLID Object-Oriented Design Principles

Do not repeat yourself (DRY)

Reuse code wherever possible.

Single Responsibility Principle (SRP)

Classes (and methods!) should have just a single responsibility

Open-Closed Principle

Objects should be open for extension, but closed for modification.

Liskov Substitution Principle

Objects should be replaceable with instances of their sub-types without affecting program correctness.

Interface Segregation Principle

Many client-specific interfaces are better than one general-purpose interface.

Dependency Inversion Principle

Depend upon abstractions, not concretions.

Detecting Problem Areas with Metrics

Supporting day-to-day activities as a developer

Prioritise tasks

- “Organise the backlog”

Convey progress on a task

- “Measure the velocity”

Evaluate whether a task is complete

- “Present work to the product owner”

Supporting day-to-day activities as a developer

Prioritise tasks

"Organise the backlog"

Convey progress on a task

"Measure the velocity"

Evaluate whether a task is complete

"Present work to the product owner"

"You can't control what you can't measure"

Tom DeMarco

Measuring Coupling and Cohesion

Fan-In	Number of other classes that depend on (i.e. have calls to) a given class. Also known as “Afferent Couplings”.
Fan-Out	Number of other classes that are targeted by calls from a given class. Also known as “Efferent Couplings”.
Response set For a Class (RFC)	The number of methods that can be called when an object of that class receives a message. Number of methods belonging to the class reachable from interface methods (can be traced in the call graph).
Lack of Cohesion of Methods (LCOM)	Value between 0 and 1, representing cohesion for class. For each data-member, count the number of methods that access it. Add up for all data members. Divide by total number of methods multiplied by total number of data members.

Chidamber, Shyam R., and Chris F. Kemerer. "A metrics suite for object oriented design." *IEEE Transactions on software engineering* 20.6 (1994): 476-493.

Measuring Size

Lines of Code LOC, KLOC, SLOC,…	Number of lines of code. SLOC - “statement lines of code” (ignore whitespace). NCLOC - “non-comment lines of code”.
Number of Methods	Count the number of methods (regardless of visibility in the code).
Number of Data Members	Number of data members (regardless of visibility in the code).

Chidamber & Kemerer's OO Metrics

Chidamber, Shyam R., and Chris F. Kemerer. "A metrics suite for object oriented design." *IEEE Transactions on software engineering* 20.6 (1994): 476-493.

Weighted Methods per Class (WMC)	Sum of method complexity scores for the methods in the class.
Depth of Inheritance Tree	Number of possible parent (and grand-parent) classes that could possibly affect a given class.
Number of Children	Number of immediate subclasses - how many classes are potentially going to inherit features from this class.
Coupling between Object Classes	Number of other classes to which a class is coupled. Coupling occurs when a method in one class uses methods or accesses data in another class.
Response for Class	Number of methods that can be invoked by sending a message to an object of the class. All public methods, inherited public methods, and all methods directly called by these methods.
Lack of Cohesion in Methods (LCOM)	Number of pairs of methods that access at least one identical instance variable.

Code Churn

Code Churn: Amount of change taking place over time.

For every new version, “churn” is the total number of added or changed lines of code over a given period of time.

Code Churn

Code Churn: Amount of change taking place over time.

For every new version, “churn” is the total number of added or changed lines of code over a given period of time.

Use of Relative Code Churn Measures to Predict System Defect Density

Nachiappan Nagappan*
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
nnagapp@ncsu.edu

Thomas Ball
Microsoft Research
Redmond, WA 98052
tball@microsoft.com

Code Churn

Code Churn: Amount of change taking place over time.

For every new version, “churn” is the total number of added or changed lines of code over a given period of time.

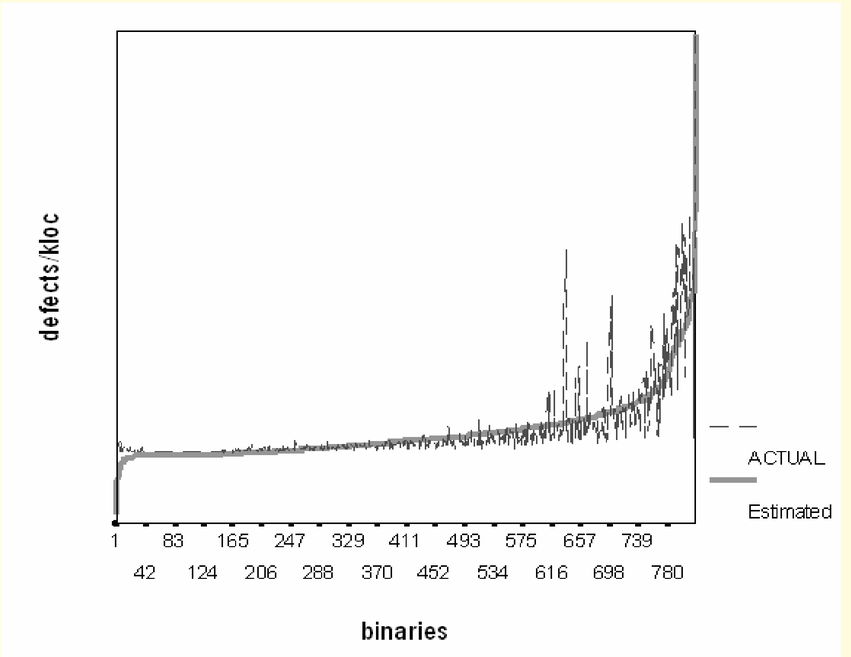
The baseline used for measuring the code churn and other measures described below is Windows Server 2003 (W2k3). We measured churn between this baseline and Windows Server 2003 Service Pack 1 (W2k3-SP1). We sometimes refer to W2k3-SP1 as the “new version” of the code. Service packs are a means by which product updates are distributed[1]. Service packs contain updates for system reliability, program compatibility, security, etc. that are conveniently bundled for easy downloading.

The size of the code base analyzed is 44.97 *million* LOC (44,970 KLOC). This consisted of 2465 binaries which were compiled from 96,189 files. Some files contribute to more than one binary. As defects for W2k3-SP1 are reported at the binary level, we relate churn to defects at the level of binaries.

Use of Relative Code Churn Measures to Predict System Defect Density

Nachiappan Nagappan*
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
nnagapp@ncsu.edu

Thomas Ball
Microsoft Research
Redmond, WA 98052
tball@microsoft.com



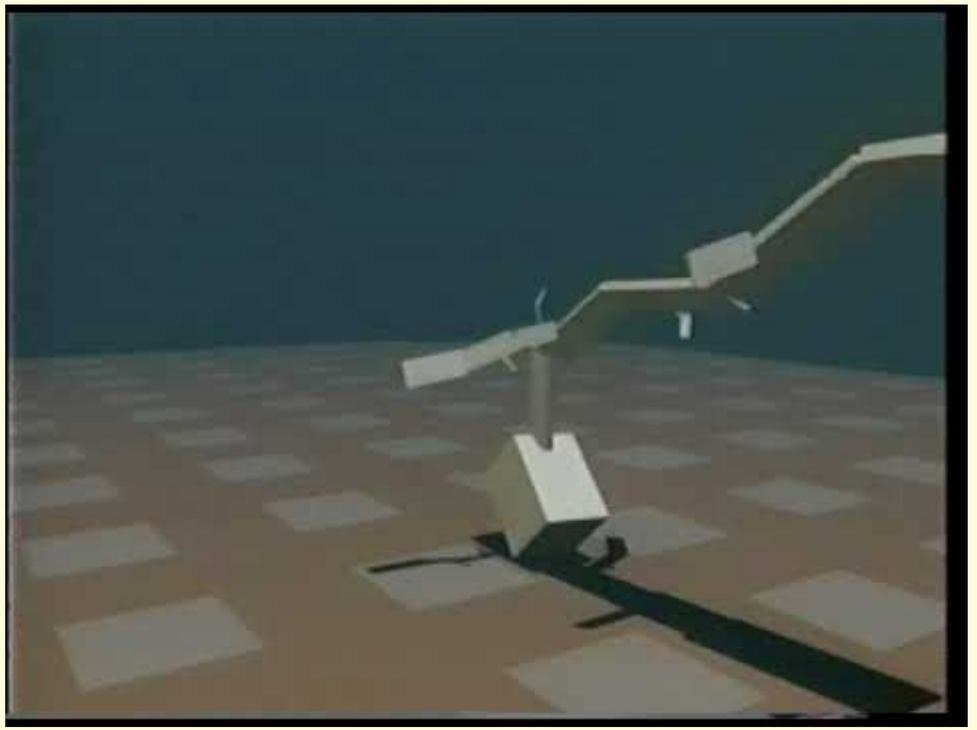
Goodhart's Law

"As soon as a measure becomes a target, it ceases to be a good measure."

Take care with metrics.

Do not use as targets.

Only use to inform as part of a bigger picture.



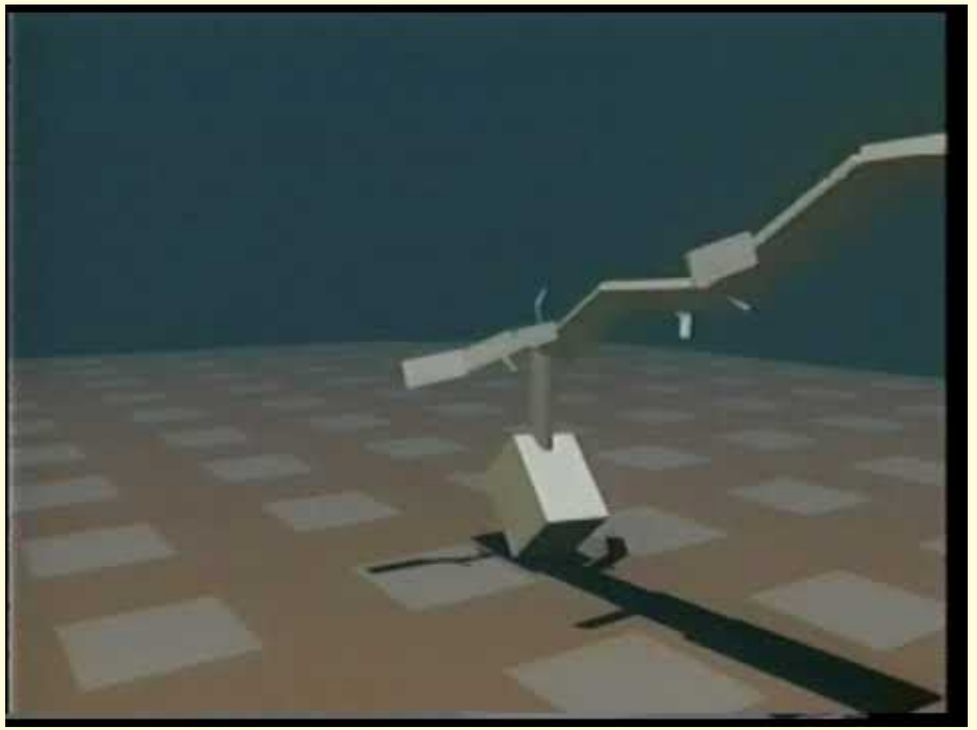
Goodhart's Law

"As soon as a measure becomes a target, it ceases to be a good measure."

Take care with metrics.

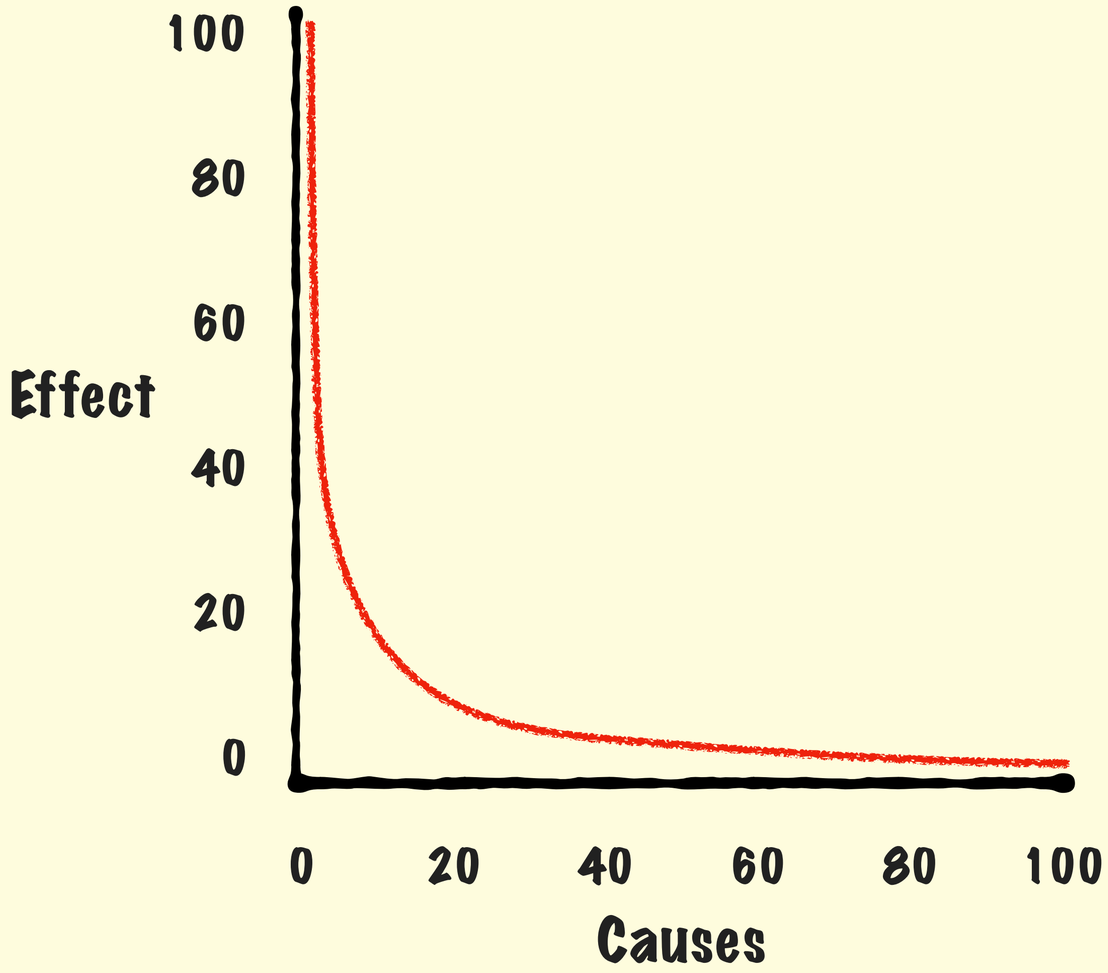
Do not use as targets.

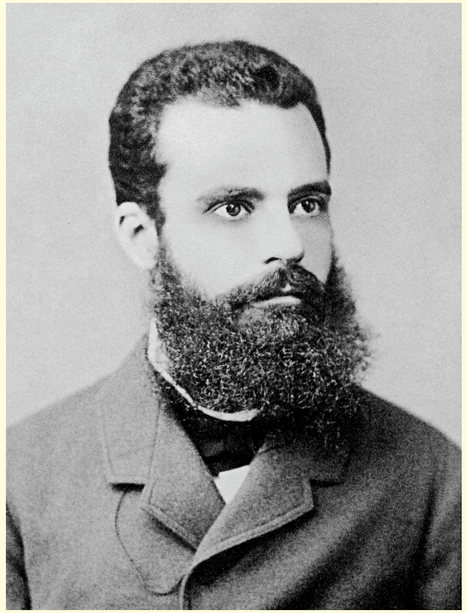
Only use to inform as part of a bigger picture.



Metrics in Practice

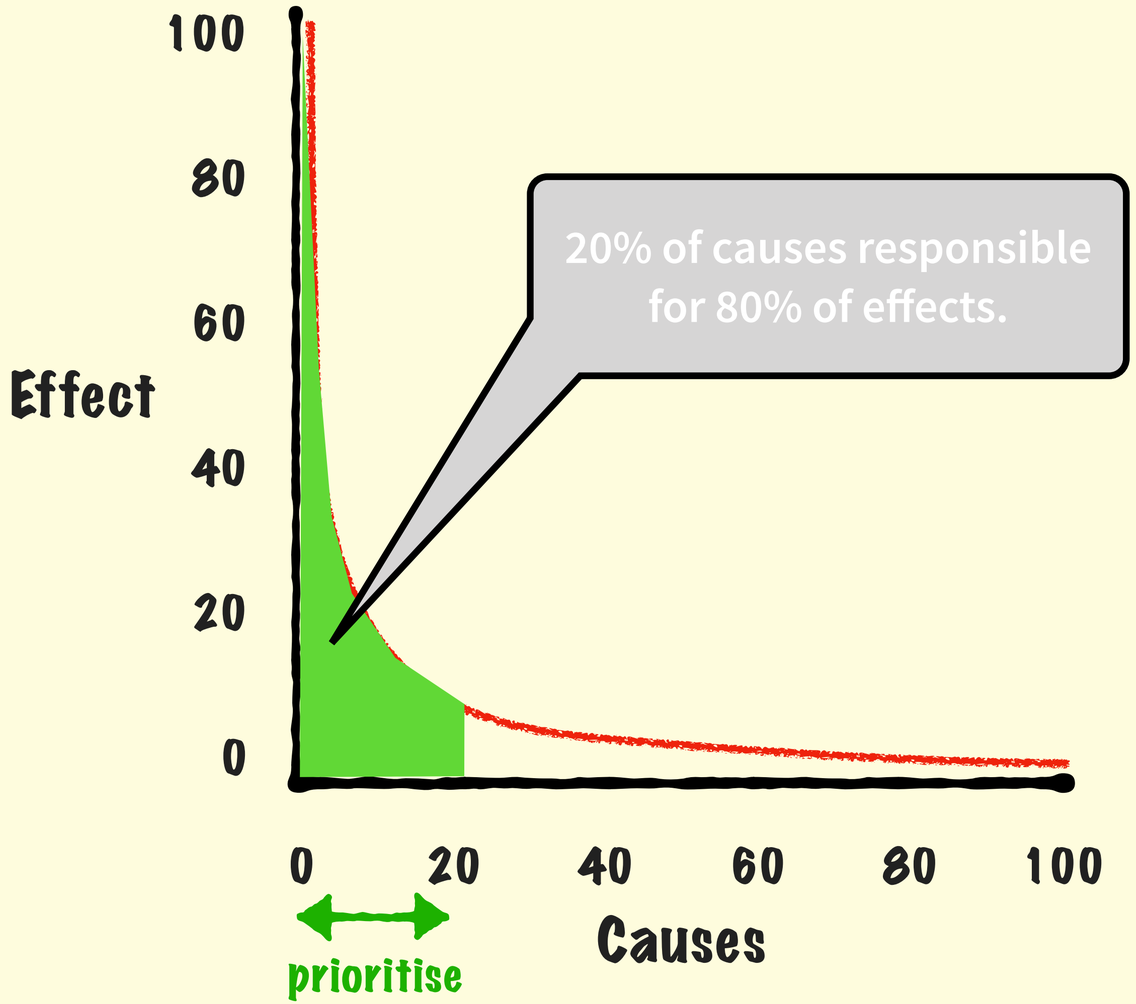
The Pareto Principle

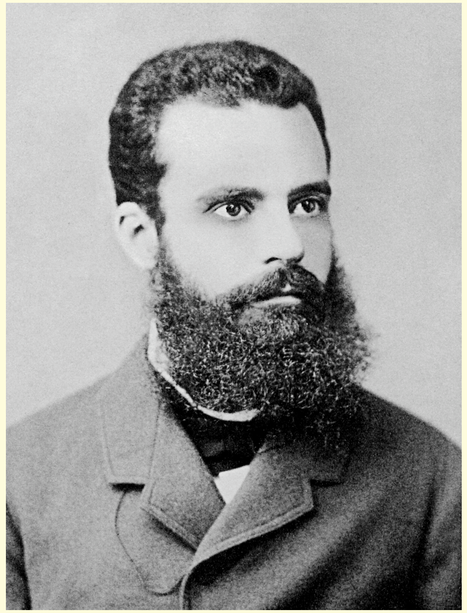




Vilfredo Pareto (1848 - 1923)

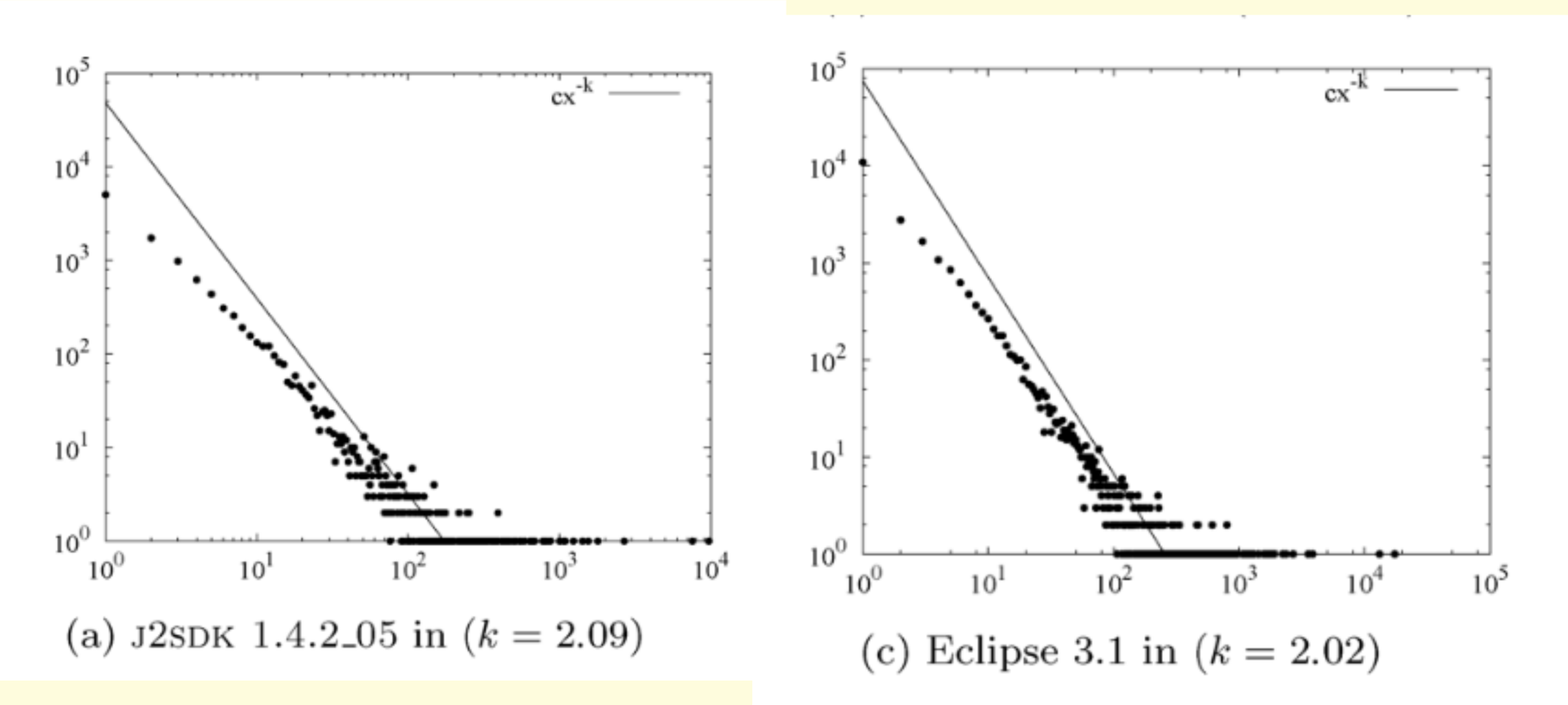
The Pareto Principle





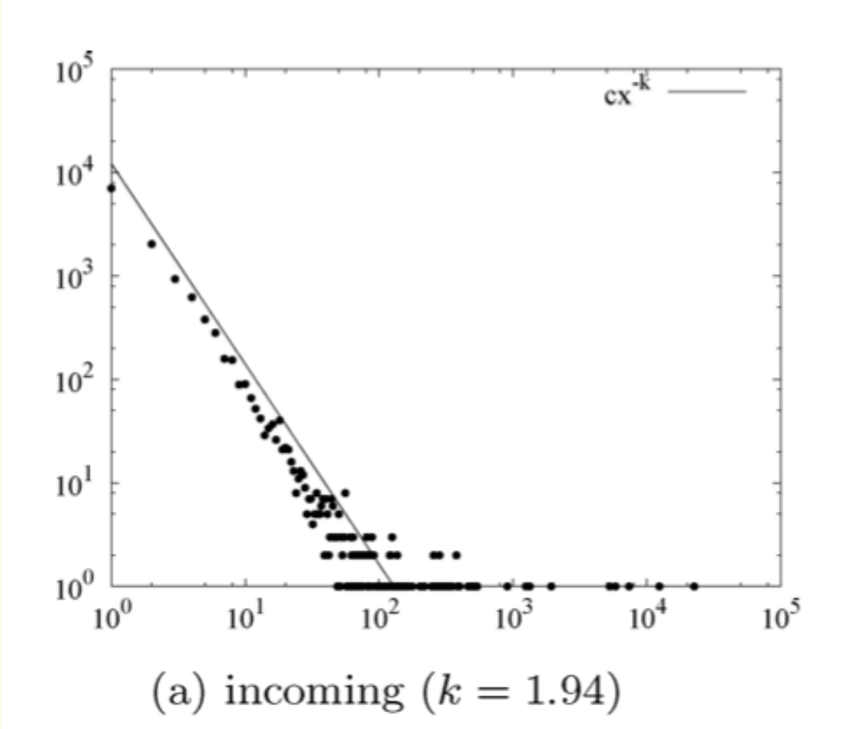
Vilfredo Pareto (1848 - 1923)

Java Dependencies



Louridas et al., "Power Laws In Software", ACM Transactions on Software Engineering and Methodology, Vol. 18, No. 1, 2008

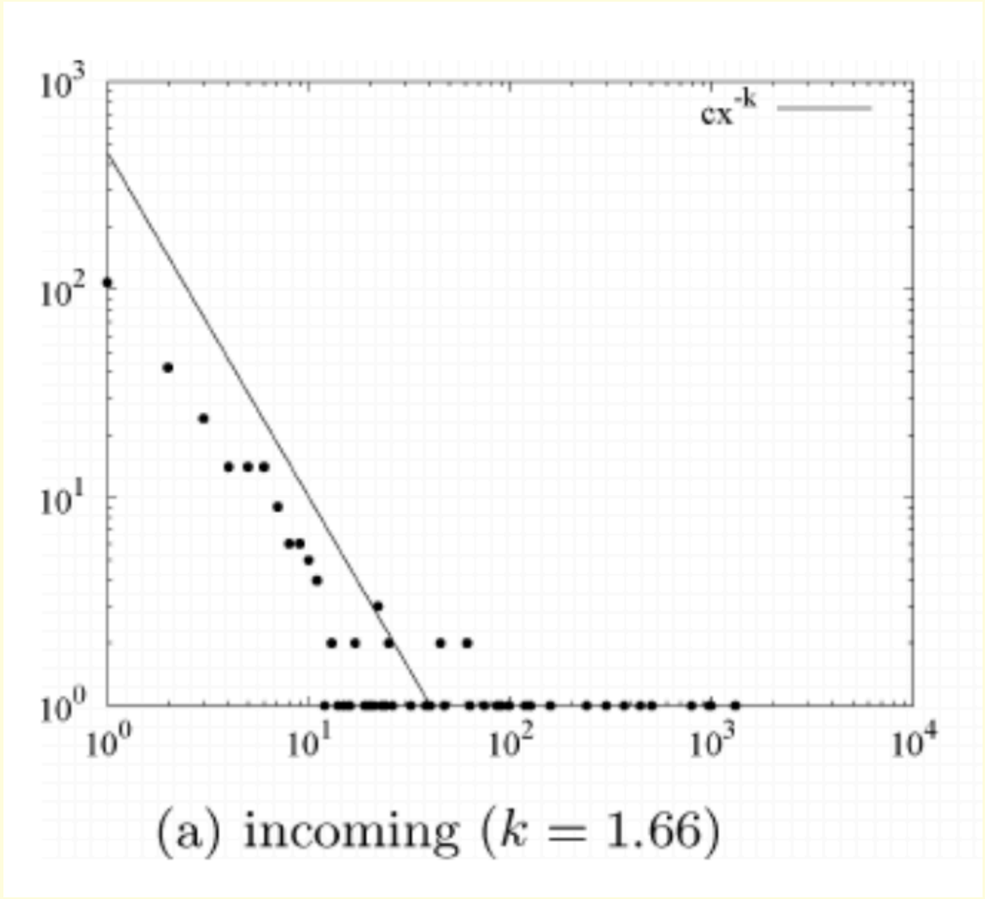
Pearl Package Dependencies



(a) incoming ($k = 1.94$)

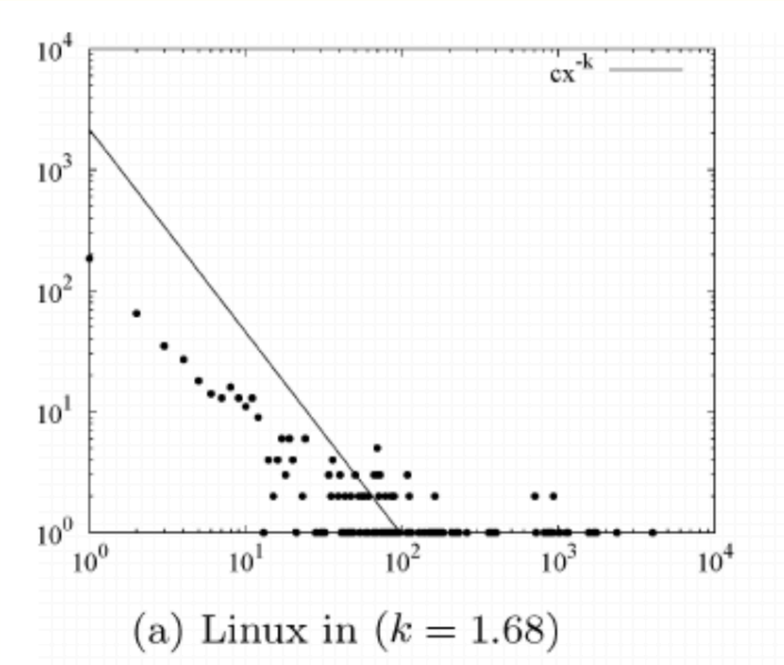
Louridas et al., "Power Laws In Software", ACM Transactions on Software Engineering and Methodology, Vol. 18, No. 1, 2008

Dependencies between Windows binaries

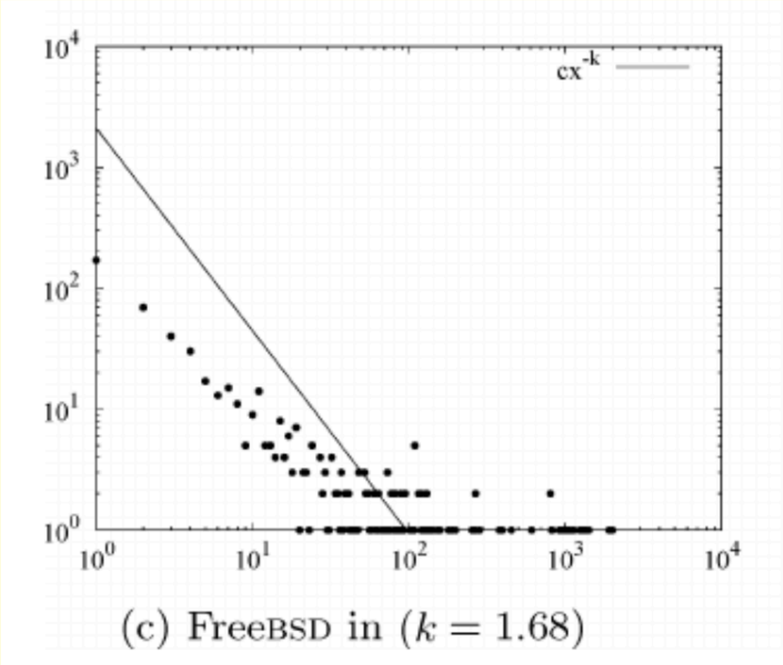


Louridas et al., "Power Laws In Software", ACM Transactions on Software Engineering and Methodology, Vol. 18, No. 1, 2008

Unix Library Dependencies



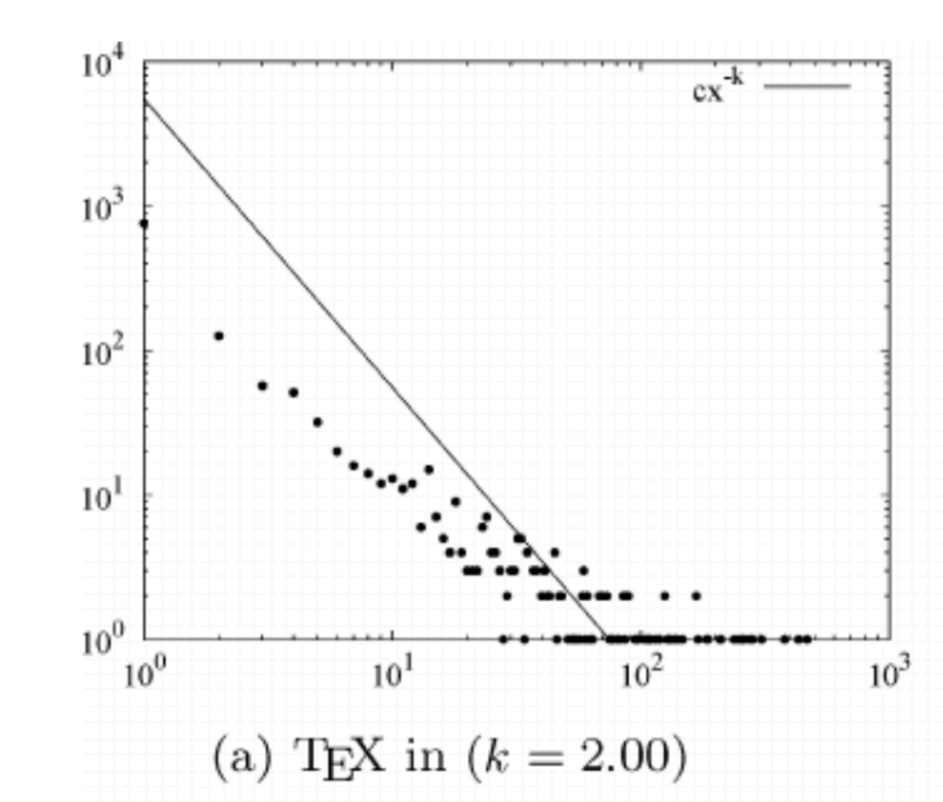
(a) Linux in $(k = 1.68)$



(c) FreeBSD in $(k = 1.68)$

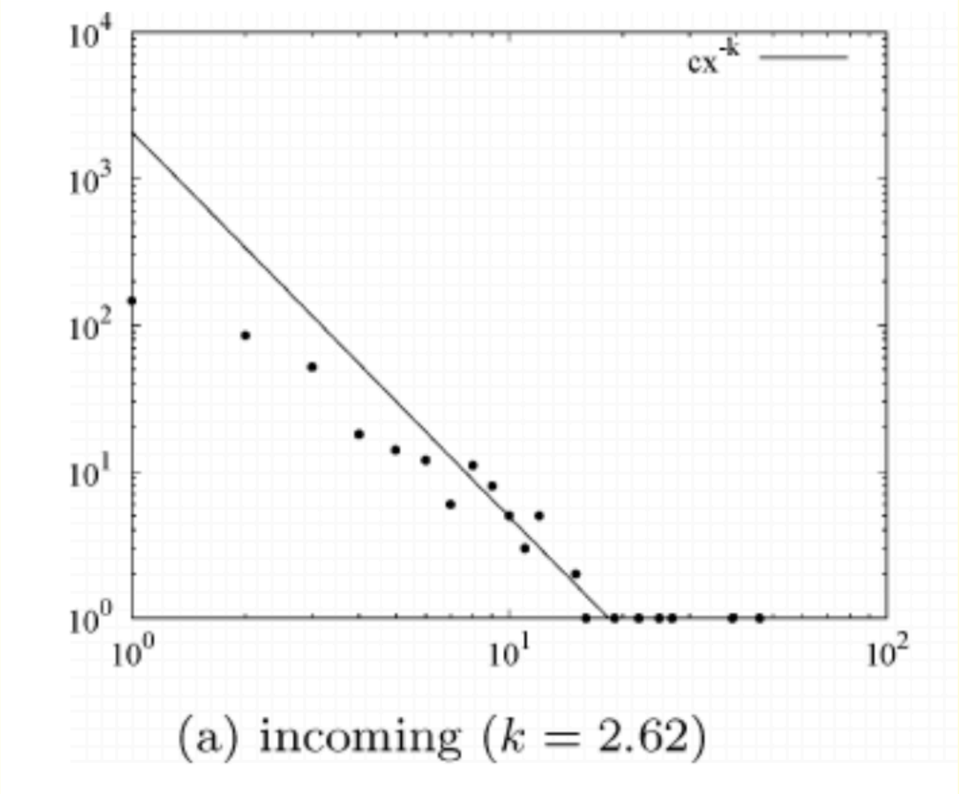
Louridas et al., "Power Laws In Software", ACM Transactions on Software Engineering and Methodology, Vol. 18, No. 1, 2008

TeX Dependencies



Louridas et al., "Power Laws In Software", ACM Transactions on Software Engineering and Methodology, Vol. 18, No. 1, 2008

Ruby Dependencies

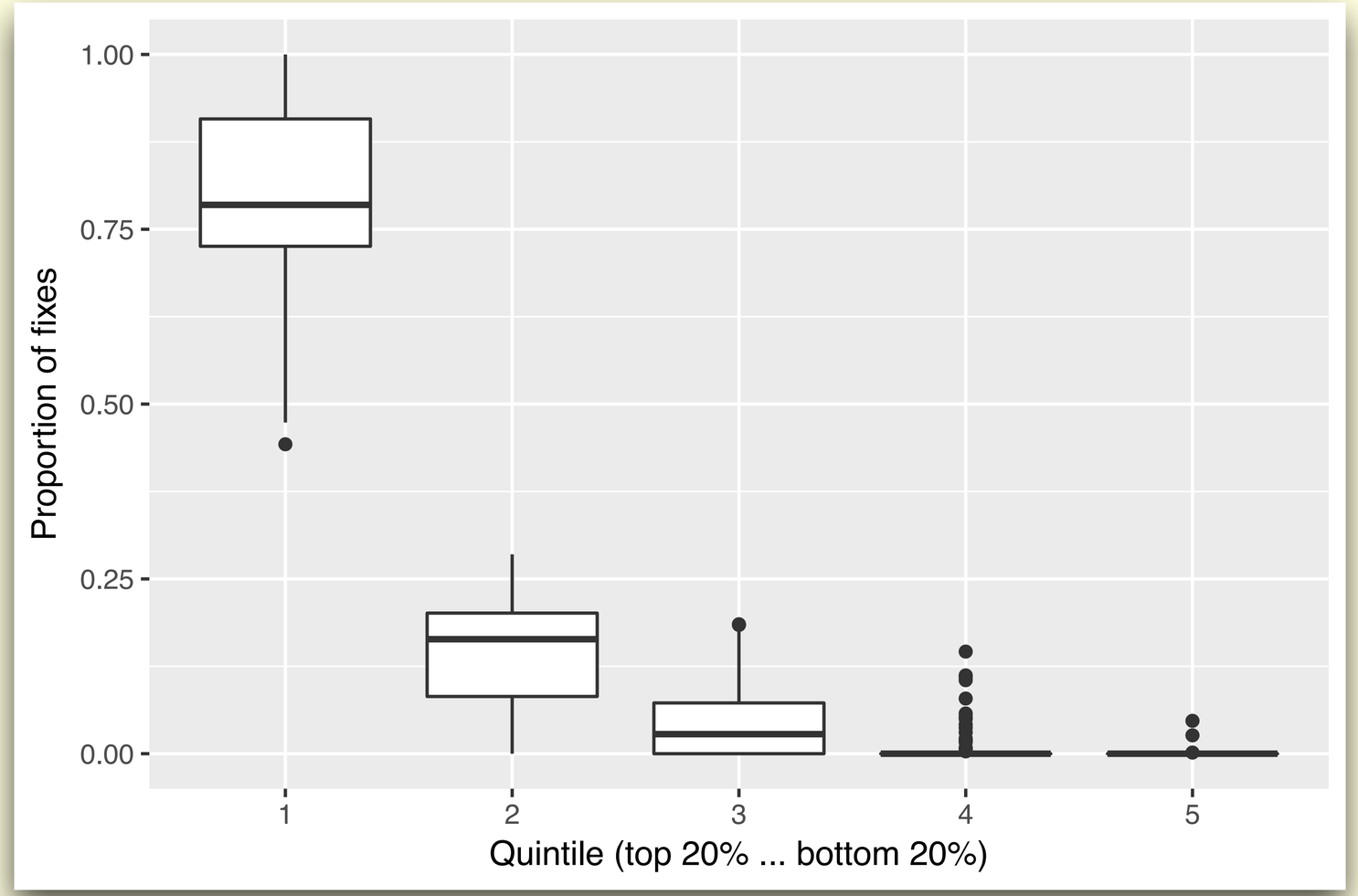


Louridas et al., "Power Laws In Software", ACM Transactions on Software Engineering and Methodology, Vol. 18, No. 1, 2008

Defects in Files

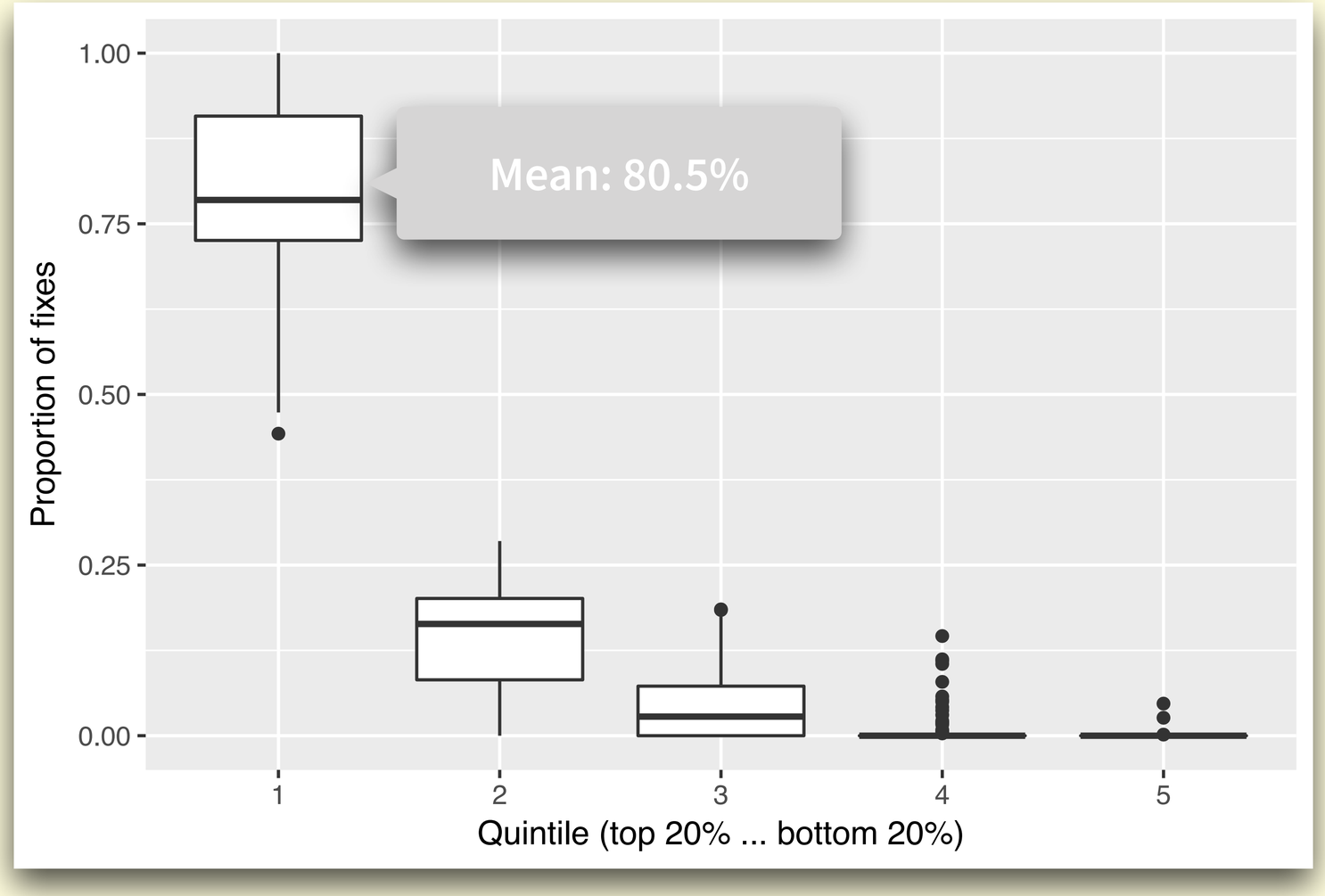
Walkinshaw and Minku, Are 20% of files responsible for 80% of defects? In Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM ’18).

Defects in Files



Walkinshaw and Minku, Are 20% of files responsible for 80% of defects? In Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '18).

Defects in Files



Walkinshaw and Minku, Are 20% of files responsible for 80% of defects? In Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '18).

Start by focussing on the top 20% (of whatever metric you're using).

Diagnosing Design Problems

Proximity of Data and Behaviour

Each class represents a single domain concept.

Data (class attributes) with associated “behaviour” - methods.

A key principle of Object-Oriented design.

If data is in a different class hierarchy …

… you need navigation code to access and manipulate it. This **increases coupling**.

If data is in the same hierarchy, but only used further down …

… you are unnecessarily exposing a variable to methods that don’t need it. This **decreases cohesion**.

Problems and (Frequent) Causes

Large number of data members

Large number of methods

Unintuitive abstractions

Large number of incoming calls

Lots of duplication

Large number of outgoing calls

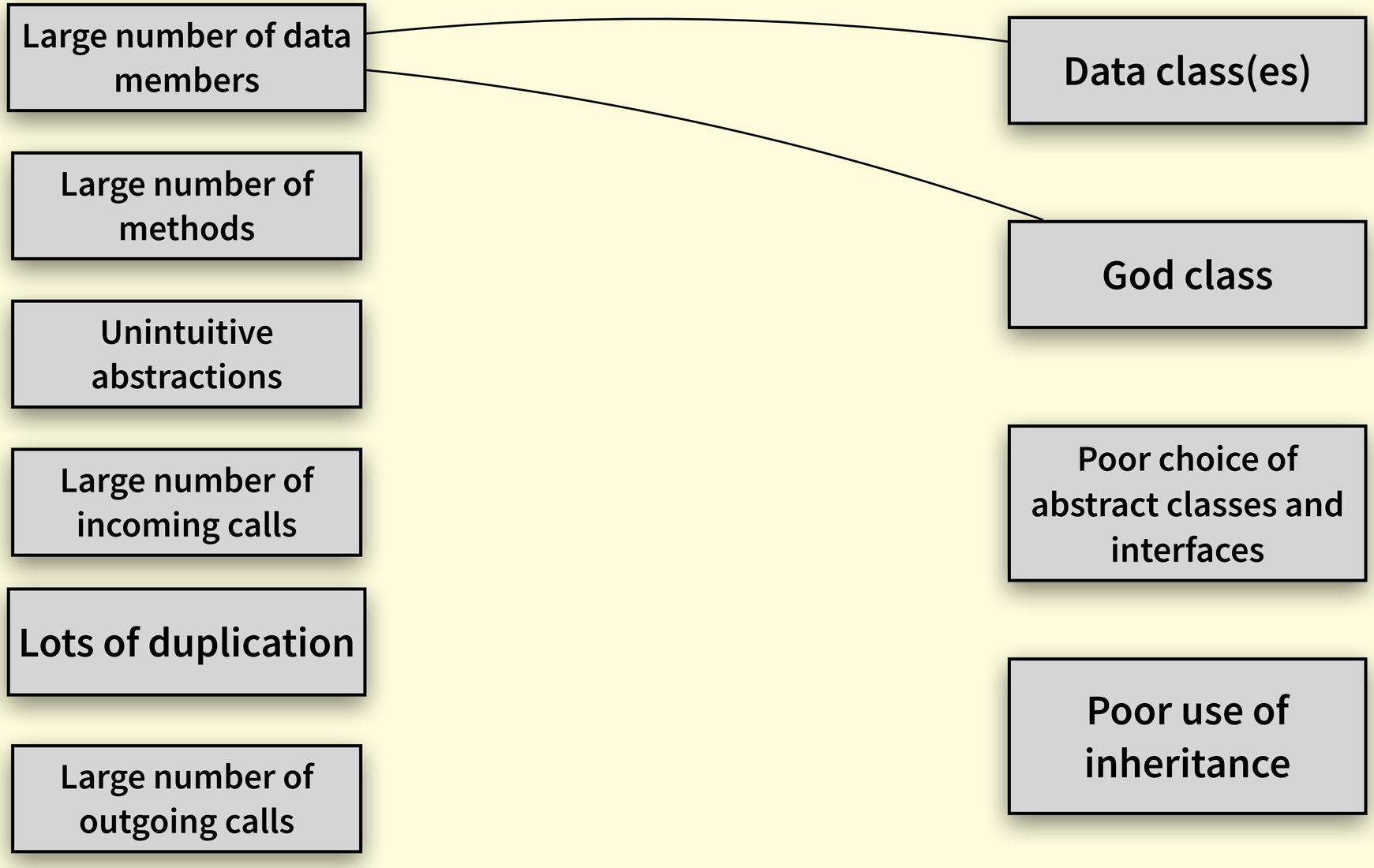
Data class(es)

God class

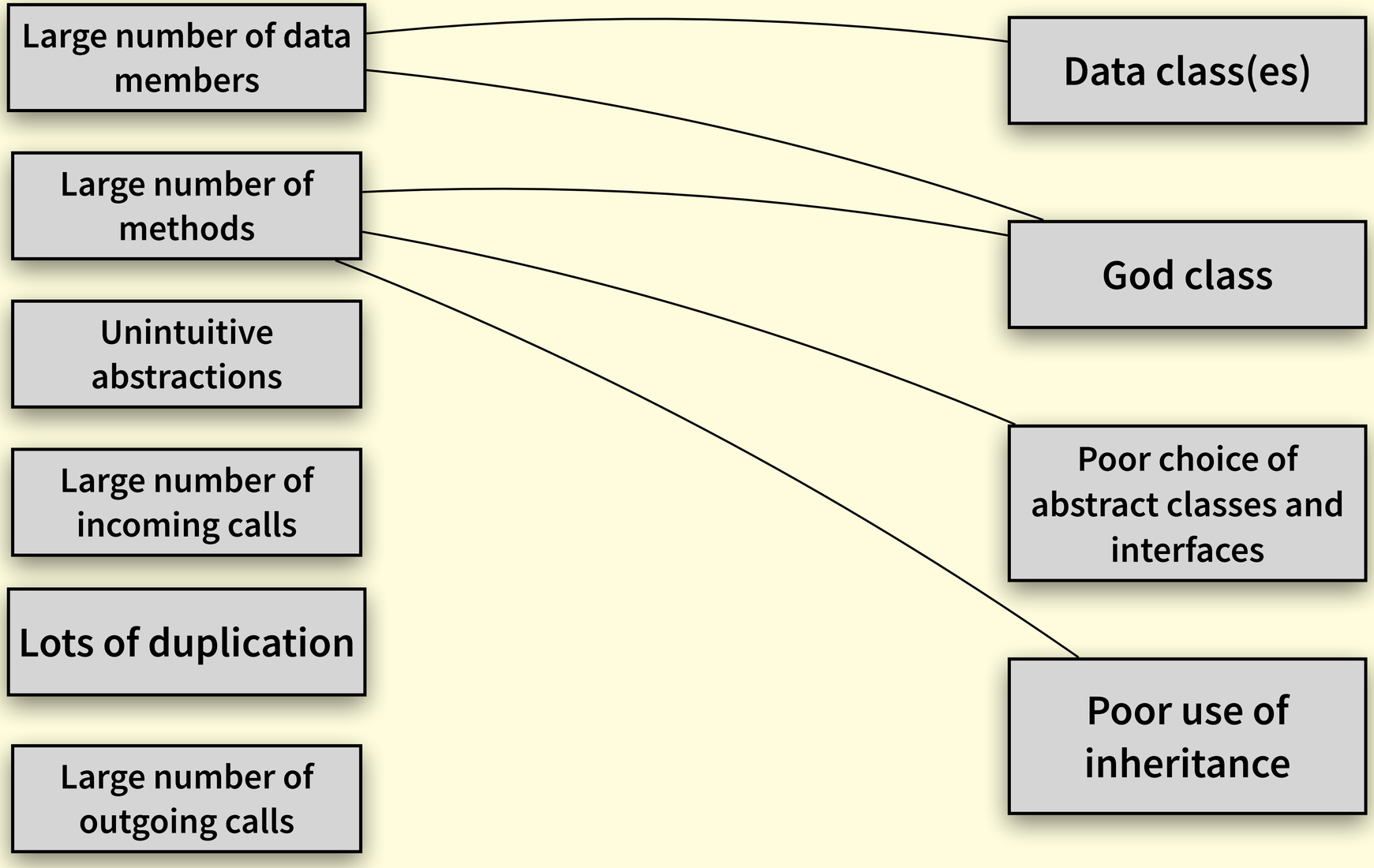
Poor choice of abstract classes and interfaces

Poor use of inheritance

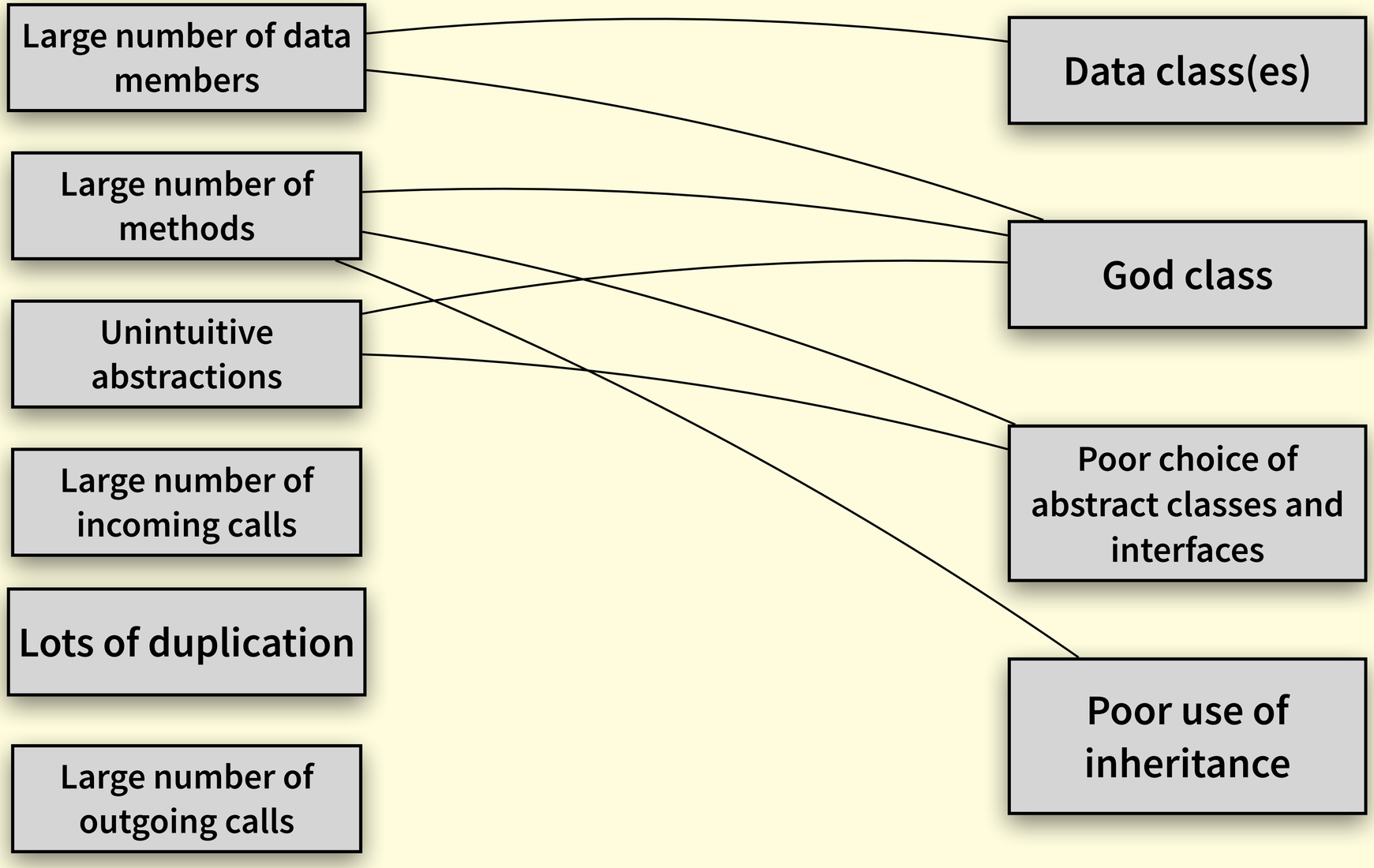
Problems and (Frequent) Causes



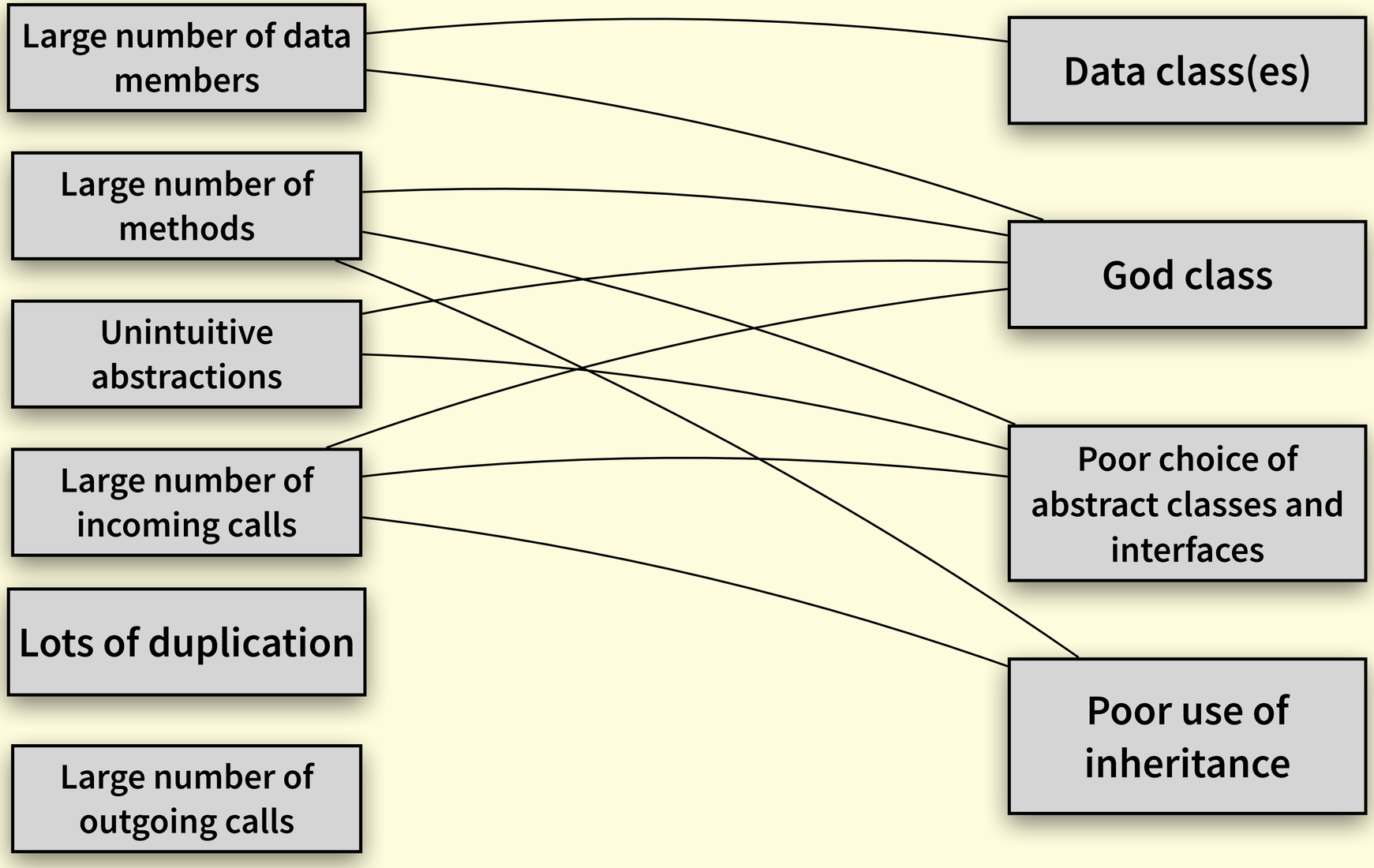
Problems and (Frequent) Causes



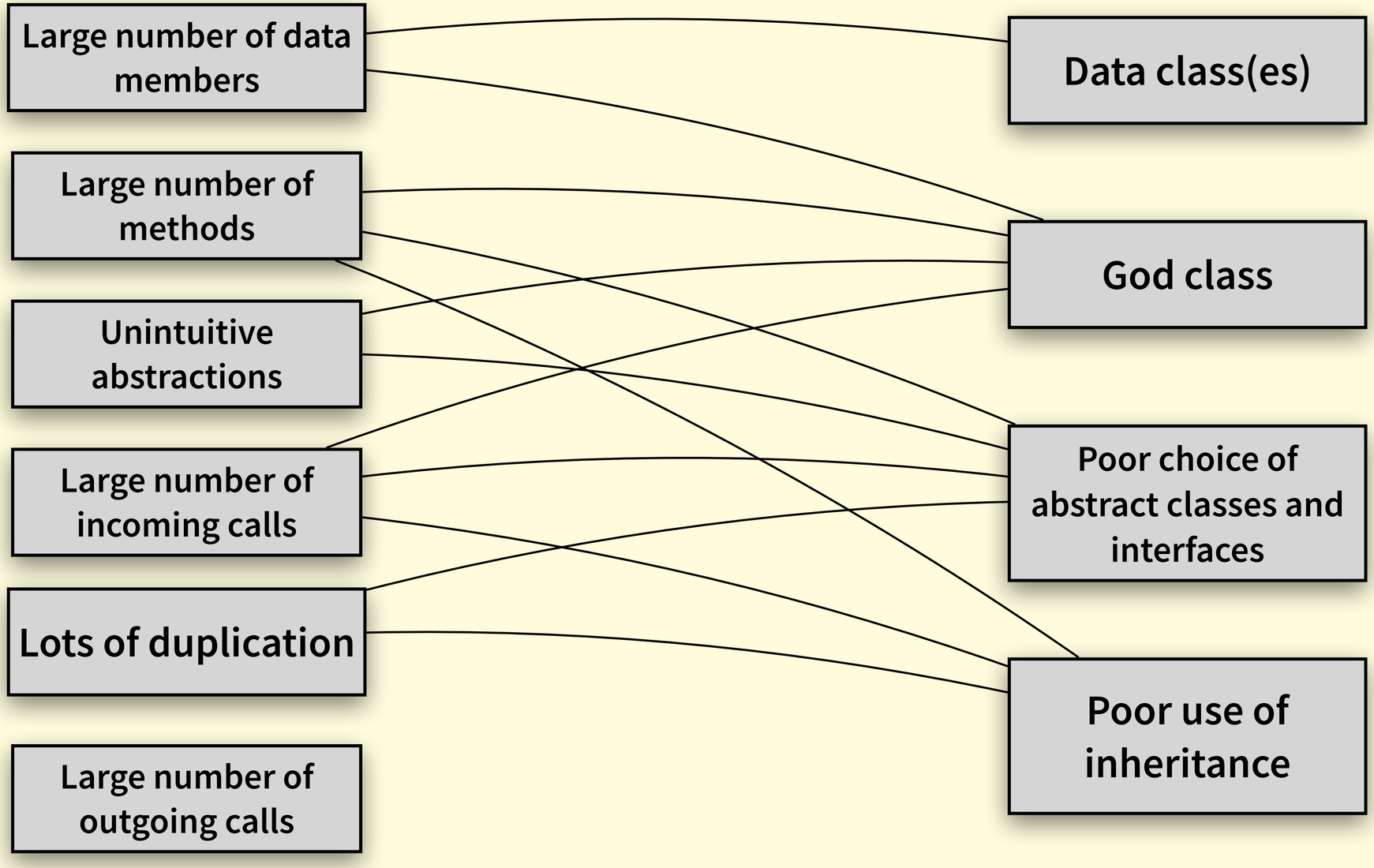
Problems and (Frequent) Causes



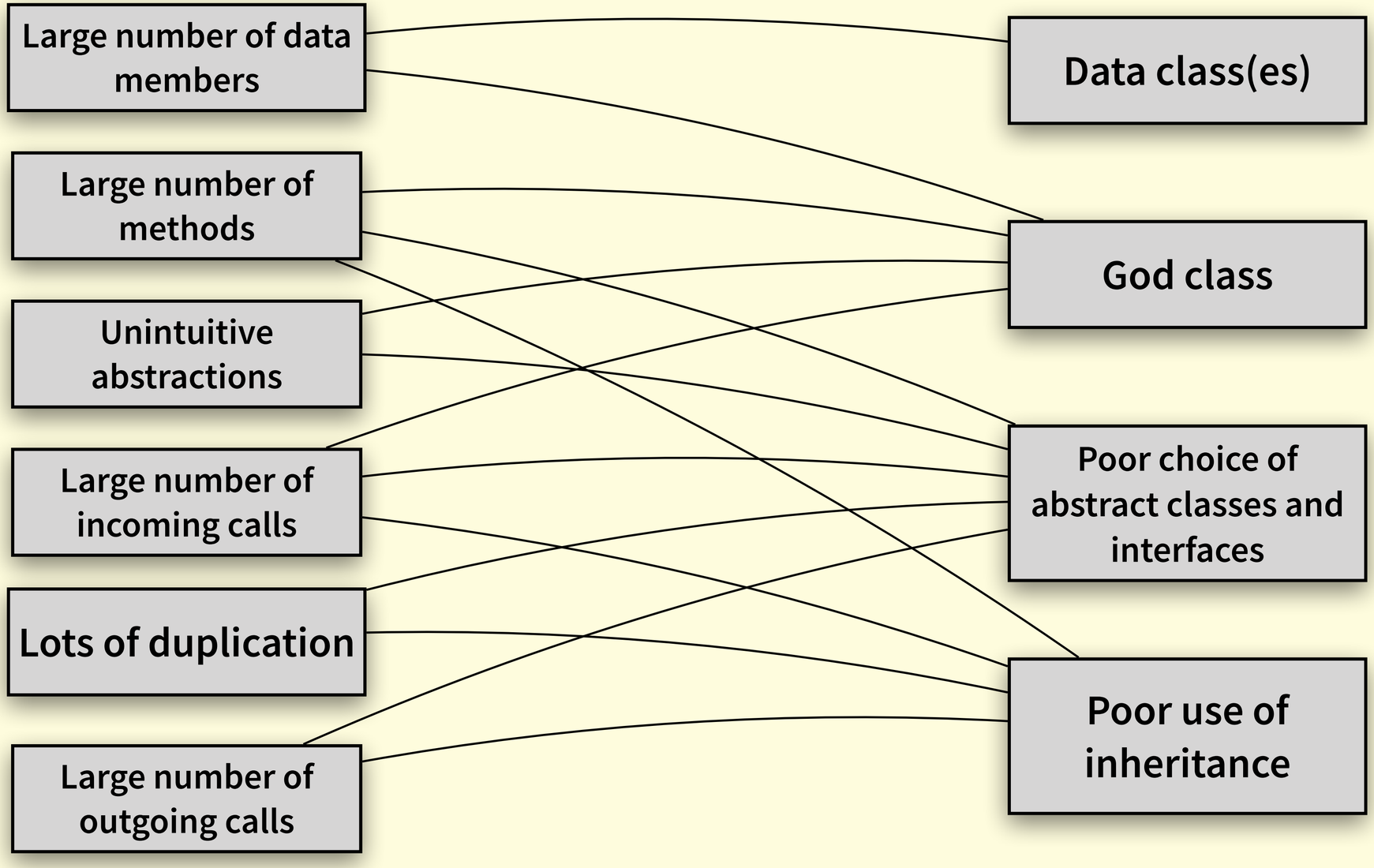
Problems and (Frequent) Causes



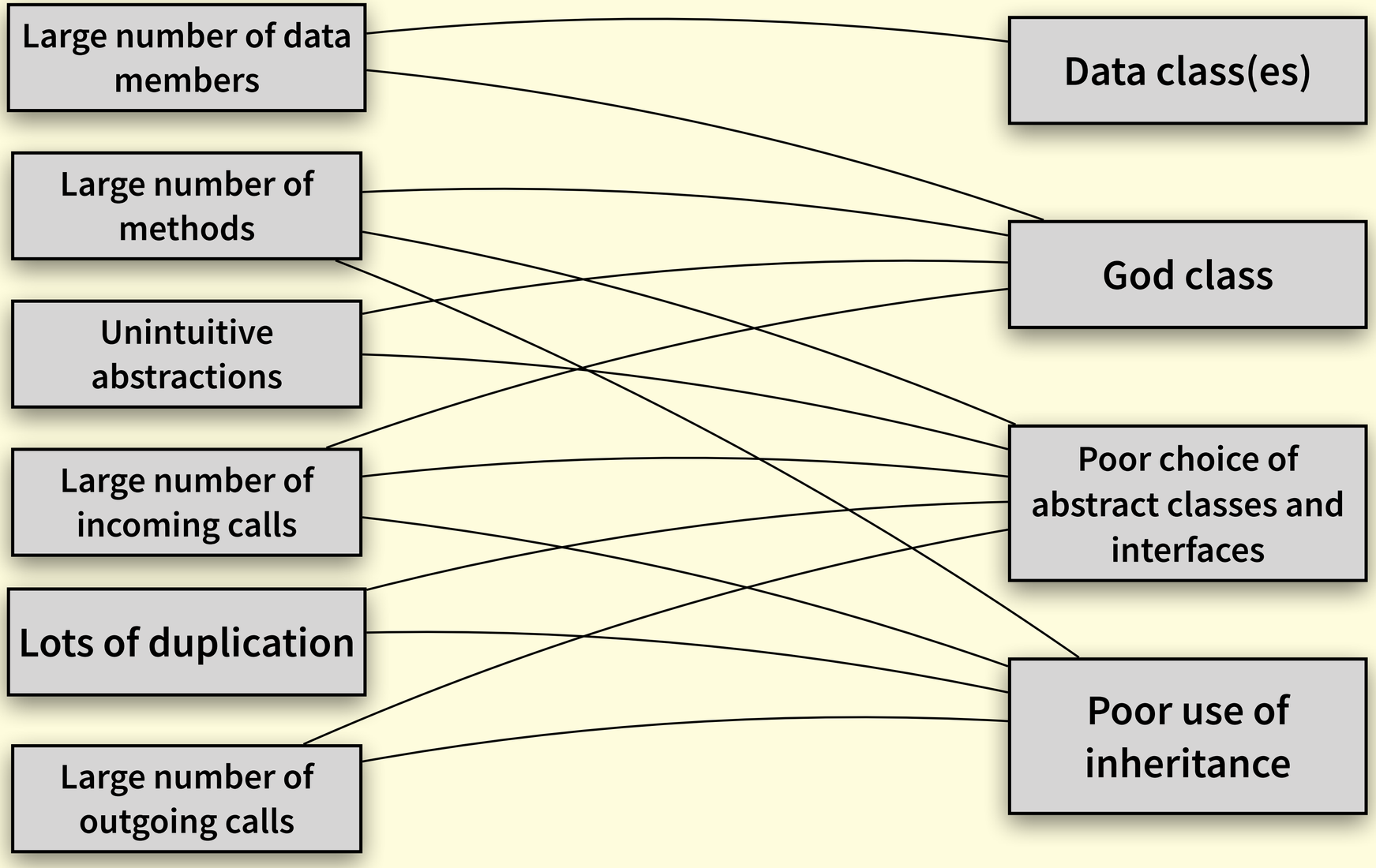
Problems and (Frequent) Causes



Problems and (Frequent) Causes



Problems and (Frequent) Causes



Misplaced Responsibilities

Key Take-Aways

The ultimate goal of reengineering is to improve design, to make it more maintainable.

Can identify key areas to home-in on by use of metrics.

- Tend to follow the 80:20 (Pareto) principle. 20% of entities are responsible for 80% of the measure.

Involves the identification of key structural design problems. Indicated by:

- Extensive coupling - fan-in, fan-out, etc.
- Excessively large classes - God Classes.
- Data classes.