Technical Debt

Thomas Sundberg

Consultant, Developer
Stockholm, Sweden
Sigma Solutions AB

thomas.sundberg@sigma.se
@thomassundberg
http://thomassundberg.wordpress.com

SIGMA

Technical Debt - Goal

Get a metaphor that can be communicated to non technical people

Know that all reasons are not technical

Know that some reasons cannot be foreseen

Have a structured refactoring method for large re-factorings

SIGMA

Disposition

Introduction

Symptoms

Reasons and solutions

A method for refactoring

Resources

SIGMA

History

Introduced by Ward Cunningham at

OOPSLA '92

Experience Report

“The WyCash Portfolio Management System”

Definition

Technical debt is a metaphor for a (technical) system where changes are (unnecessary) hard.

- New functionality
- Bug fixes
- Brittle development
- Brittle testing

SIGMA

Financial debt

Technical debt is similar to a financial debt

- Take a loan (to buy a house)
- Have to pay interest (while living in the house)
- Have to pay back the loan (when selling the house)

Technical debt: You loan time in the future

- It is more expensive than to do it correct from the beginning

It may be worth the extra cost

SIGMA

Consequences

Motivation disappears

- It's not fun anymore

Velocity goes down

- New features are harder to add
- Bugs are harder to fix

Builds up slowly

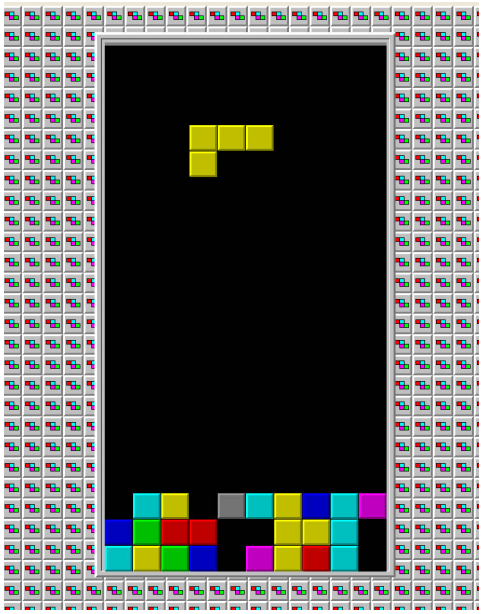
- Like hunger

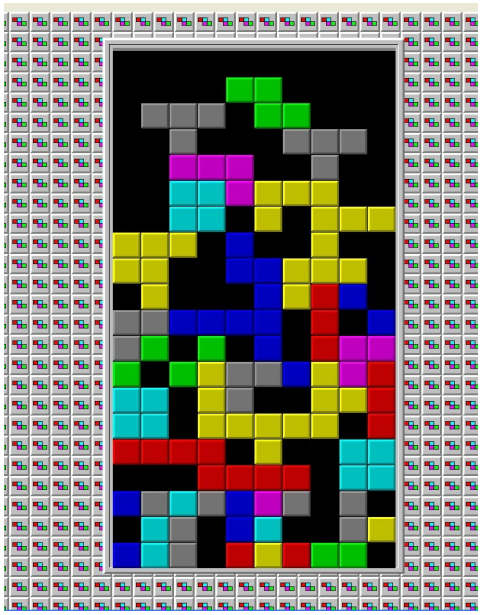
Development finally grinds to a halt

- Out of business
- Bankruptcy

Metaphor

Tetris





Game over

- Bankruptcy?

SIGMA

Metaphor

Doing the dishes

- Washing up every day

- Seldom...

SIGMA

Disposition

Introduction

Symptoms

Reasons and solutions

A method for refactoring

Resources

Gut feeling

Customers

- Everything is really slow and difficult

Developers

- This code is bad and embarrassing

SIGMA

Metrics

Method lengths

Class lengths

Velocity

Disposition

Introduction

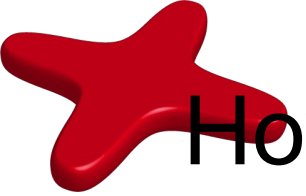
Symptoms

Reasons and solutions

A method for refactoring

Resources

SIGMA



How do you create a technical debt?

- 3rd party
- Law or Market
- Upper management
- Technical reasons

SIGMA

3rd party

Framework leakage & Vendor lock in

- External APIs all over the code
- No abstractions to keep external APIs away from the core solution

Abstractions

- Hide 3rd party API behind your own layer
- Push external code away from your domain

SIGMA

Law or Market

- New laws
- New competition
  - Hard to prepare for
    - Hard to foresee the future
      - If you can, change occupation
- Prepare for any changes
  - Automate testing
  - Fast deploy and un-deploy
  - Frequent releases
  - Accept changes as a part of life

SIGMA

(Upper) Managment

Decisions will impact the technical solution

Inform about the technical implications

- Talk and reason about it

SIGMA

Market window

Feature foo must be on the market at a certain date

- Or we might as well not deliver it at all

Accept if there are good reasons

Plan for how the technical debt will be re-paid

SIGMA

Insufficient domain knowledge

If you don't understand the problem you need to solve, how can you solve it?

Make sure that the developers are educated in the domain

Visit the customer and end users and see how they work

SIGMA

Lack of information

Incorrect assumptions are the mother of all f**k ups

Always verify your assumptions

State your assumptions as clear as possible

SIGMA

Lack of respect and disagreements

Developers create own rules

Break builds

Different solutions to similar problems

Communicate within the team

- Agree on team rules and stick to them
- Discuss pros and cons with different solutions to the same problem
- Choose one solution and stick to it as long as it makes sense

SIGMA

Staff turn over

It takes time to learn a new code base

Knowledge gets lost

Spread the knowledge

- Pair programming

Build automatic verification into the system

- Test Driven Development, TDD

Lead people as if they were volunteers *

* @Fredrik_Jonsson, Agila Sverige 2009

SIGMA

Stress

High pressure from

- Management
- Surrounding system

Not working in a sustainable pace

No overview of workload

Always cutting corners

- Never finish anything properly

A Tetris situation

SIGMA

Stress

- Discuss with management
  - This cause us to deliver bad result
- Change to a sustainable pace
  - 40 hour a week
- Visualise the work load
  - Introduce a taskboard
- Add slack to the system

SIGMA

Slack

Utilizing a system 100%

Results in (almost) zero throughput

SIGMA

Slack

Utilizing it to a few percent

Results in (almost) 100% throughput

SIGMA

Slack

Challenge:

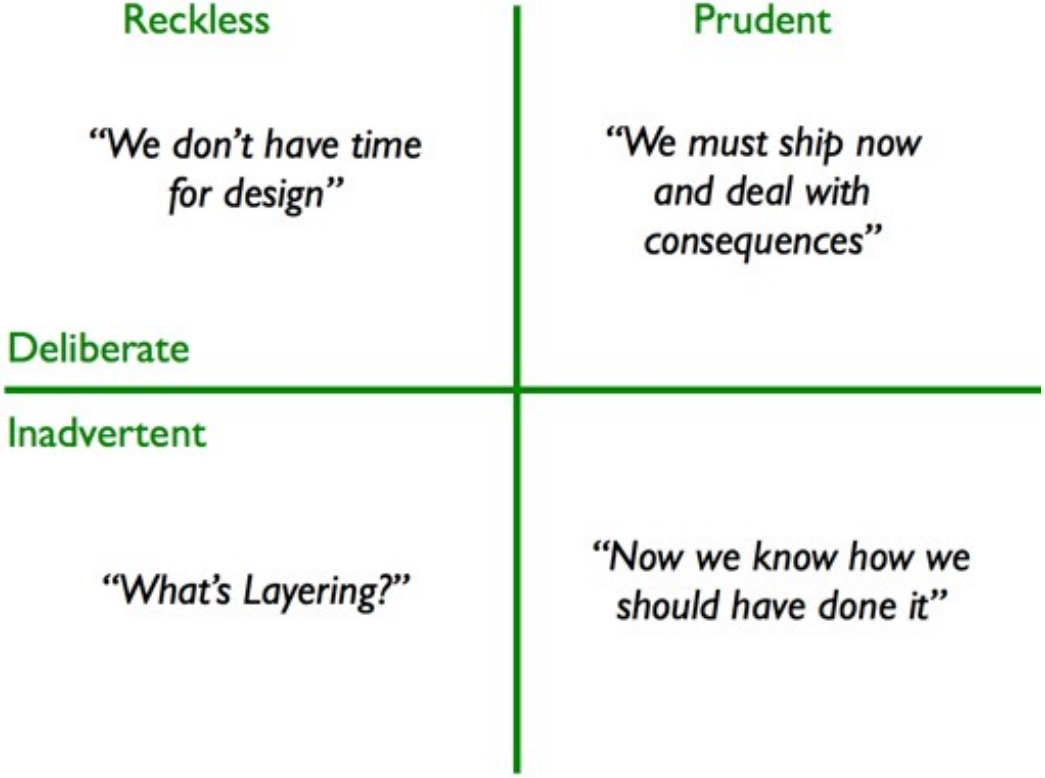
Have your developers work a week and don't tell them what to do

They will probably reduce your technical debt

SIGMA

Technical reasons

Technical debt quadrant



http://www.martinfowler.com/bliki/TechnicalDebtQuadrant.html

SIGMA

Technical debt quadrant

Reckless

- We don't care (for some reason)

Prudent

- We are cautious and takes a calculated risk

Deliberate

- We choose to do this for a reason

Inadvertent

- We don't know better

SIGMA

Insufficient technical knowledge

Education

- Study group
- Read books together
- Coding dojos

Pair programming

- Transfer knowledge

Test Driven Development, TDD

- Solve the problem twice
- At least the double quality

Clever code

The execution path is hard to follow

- Mutable state
- Conditions
- Non standard solutions
- Difficult to maintain

Avoid if possible

- Keep it simple
- The simplest possible solution is good enough

SIGMA

Premature optimization

Before testing live

- May result in an optimization of the wrong problem

Something that isn't used

- Isn't worth anything

An incorrect solution

- Worthless

Will add unnecessary complexity

SIGMA

Not invented here

Not willing to accept somebody else's solution

- Re-writing a solution that works

Accept that other people are clever and use their solutions

- Other clever developer do exist

SIGMA

Found elsewhere

Only external APIs

Complex frameworks that solves much more than needed

Add a layer between your domain code and the framework to shield yourself from it

Accept that you need to develop code

SIGMA

Inherited legacy code

Legacy code dropped in your knee

Add tests

- Whenever something need to change

Automate verification

- Automate testing before a change

SIGMA

Broken window syndrome

Living with small errors will slowly drive you into accepting larger errors

- You will build technical debt

Fix small things before they grow up to large things

- Remember the dishes?

Disposition

Introduction

Symptoms

Reasons and solutions

A method for refactoring

Resources

Refactoring

- It's the most important technical tool you have
  - Change your code without changing it's behaviour
  - Use good tools
  - A version control system
    - Git or Mercurial are trivial to get started with
  - Use them properly
- Make room for future functionality

Refactoring

A simple case

- Run all tests
- Change the name of a method
- Run all tests
- Checkin if the test results are the same
- You have decreased the technical debt

A trivial example with one important point

- Always run your tests first
- Always run them again before checkin so you know if you broke anything

SIGMA

A large refactoring

A large refactoring is (almost) the same as a small refactoring

- May take longer time
- May need many subgoals
- Avoid if possible

A refactoring example

We shall refactor a central part of a large system

Many dependencies will be affected

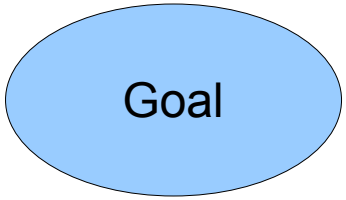
It will take a long time

The system must work all the time

How can this be done?

A dependency graph

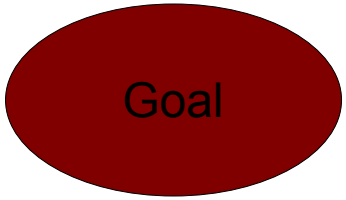
Start with the goal



SIGMA

A dependency graph

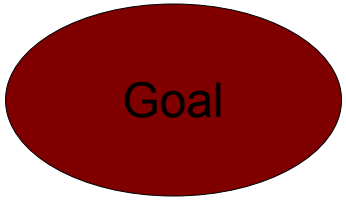
Fix it with the simplest solution that can work



SIGMA

A dependency graph

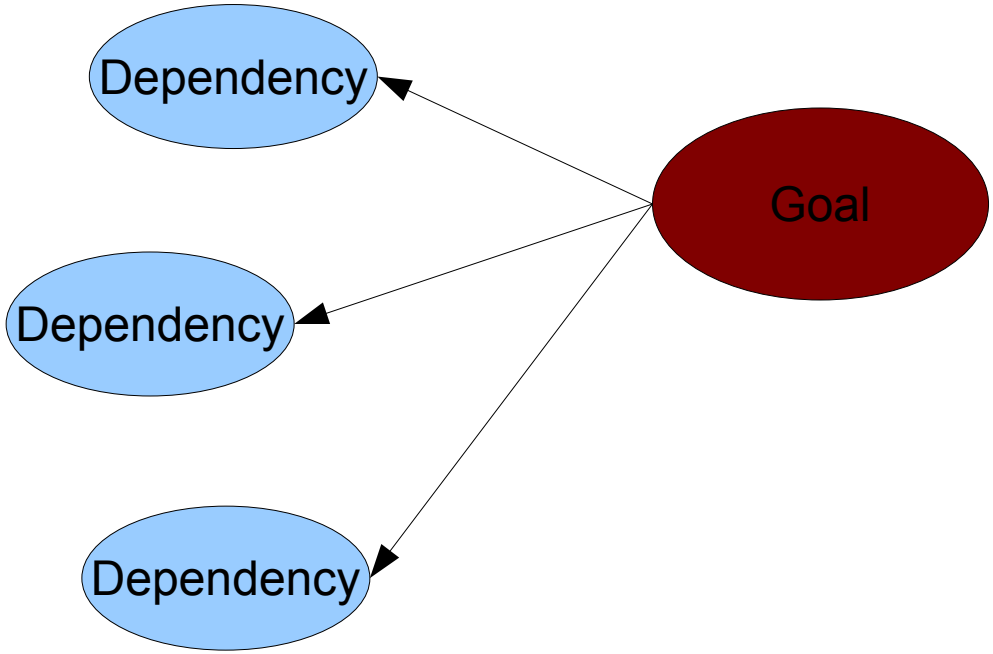
Run all tests



SIGMA

A dependency graph

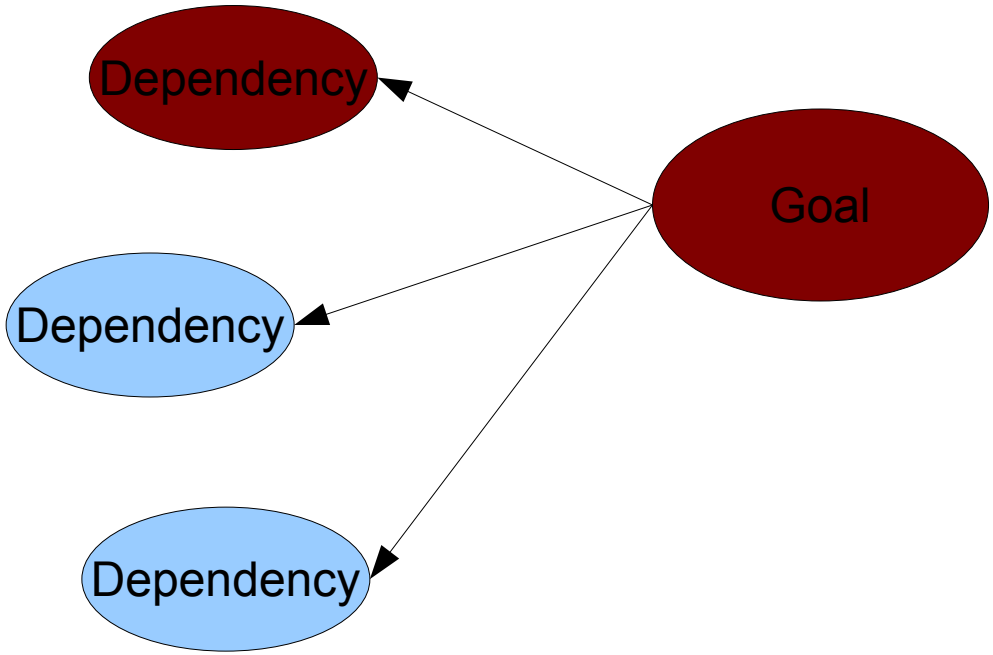
We found three dependencies that was affected by the change



SIGMA

A dependency graph

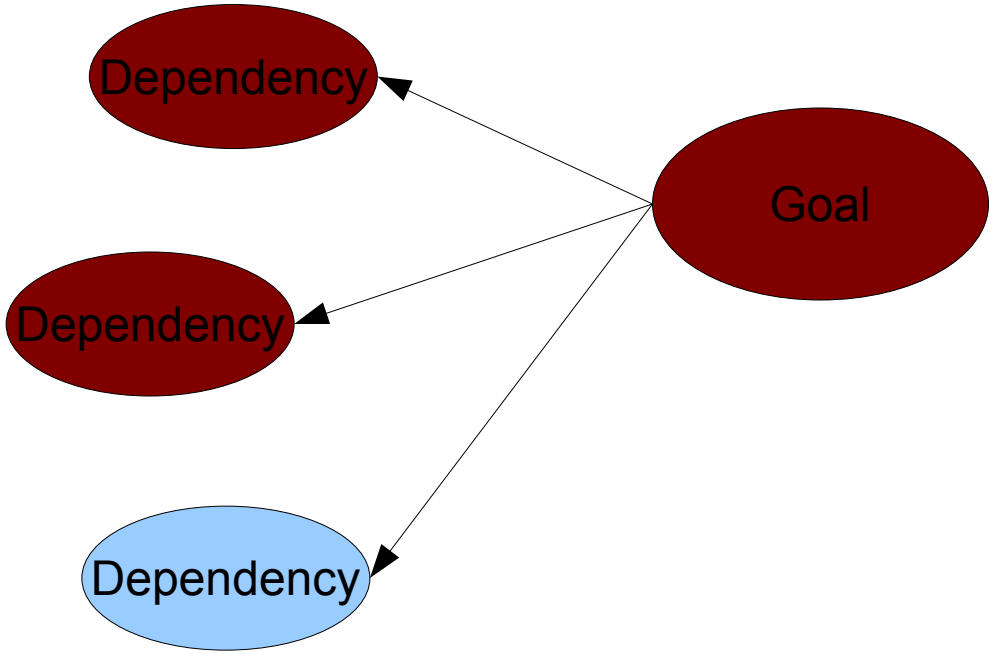
Fix each of the dependency



SIGMA

A dependency graph

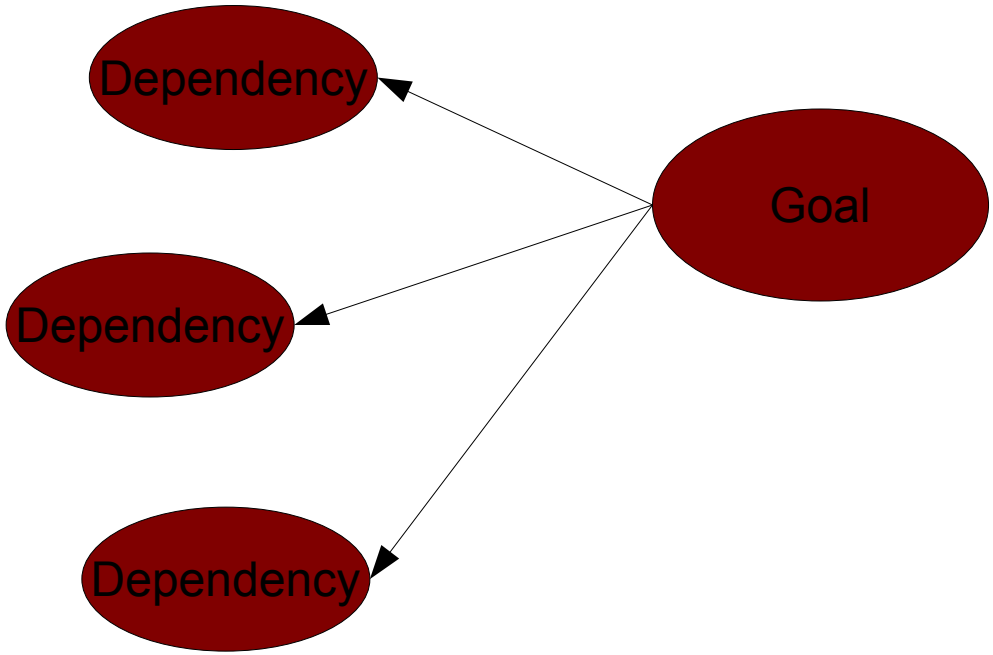
Fix each of the dependency



SIGMA

A dependency graph

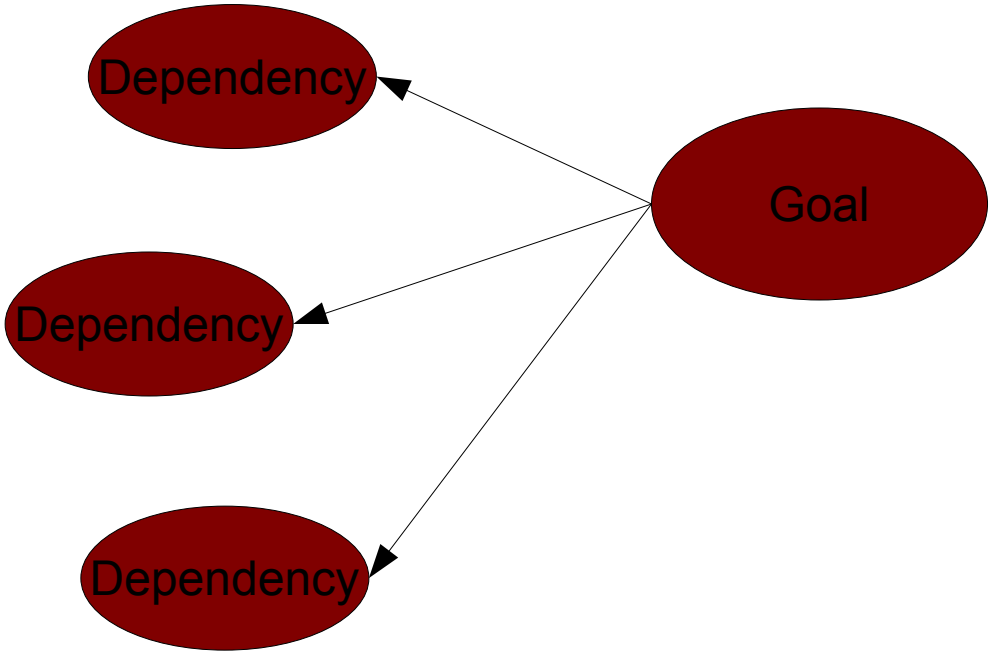
Fix each of the dependency



SIGMA

A dependency graph

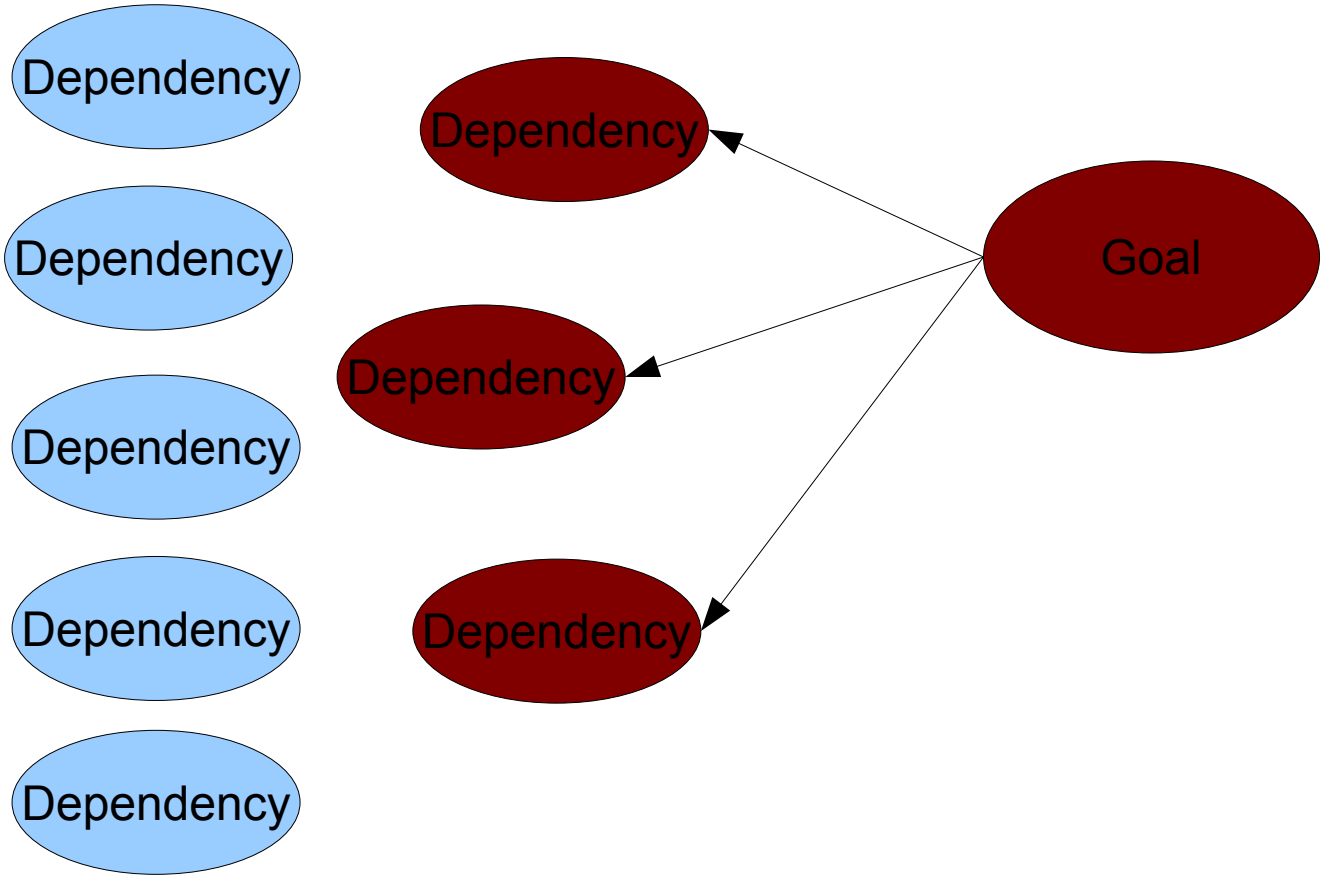
Run all tests again



SIGMA

A dependency graph

We found more dependencies



SIGMA

A structured refactoring method

We can fix all new dependencies that we found

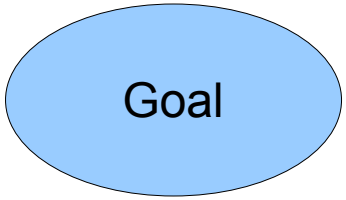
- It may spread like rings on water
- Hard to tell how long time it will take

How can we make sure that the system works all the time?

A new approach is needed

A structured refactoring method

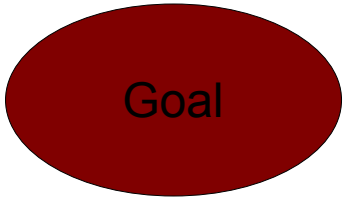
Start with the goal



SIGMA

A structured refactoring method

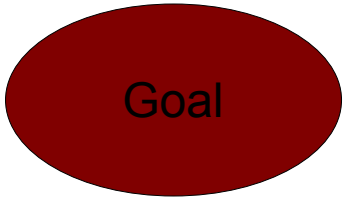
Fix it with the simplest solution that can work



SIGMA

A structured refactoring method

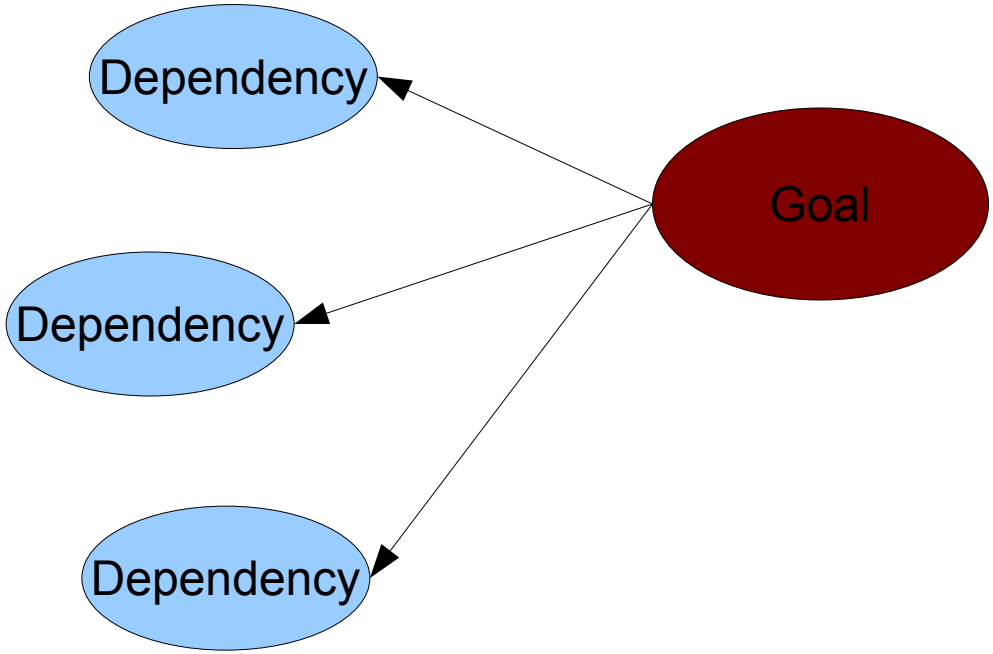
Run all tests



SIGMA

A structured refactoring method

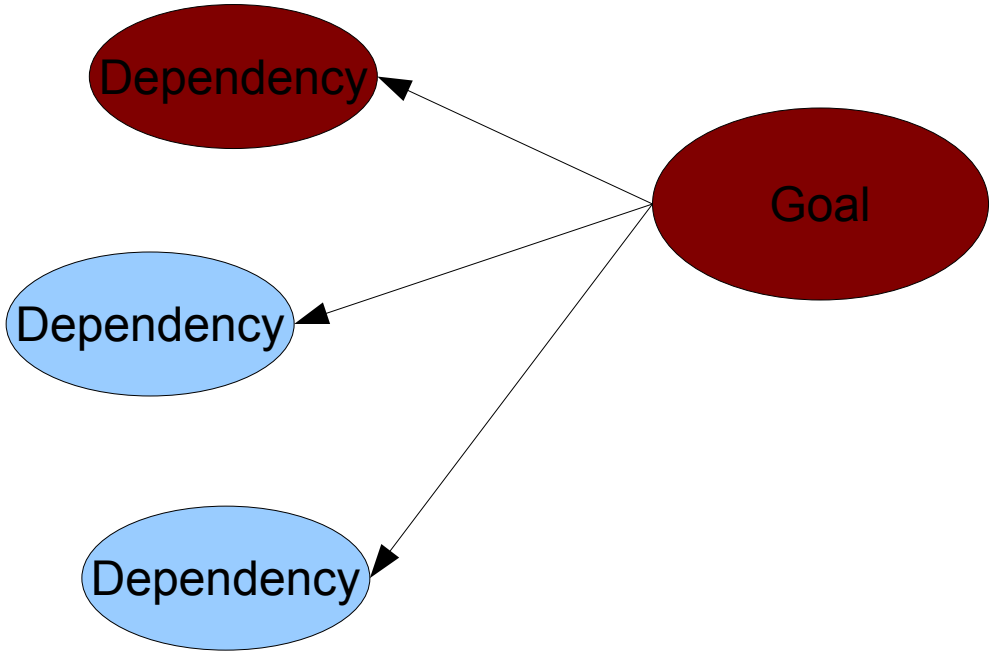
We found three dependencies that was affected by the change



SIGMA

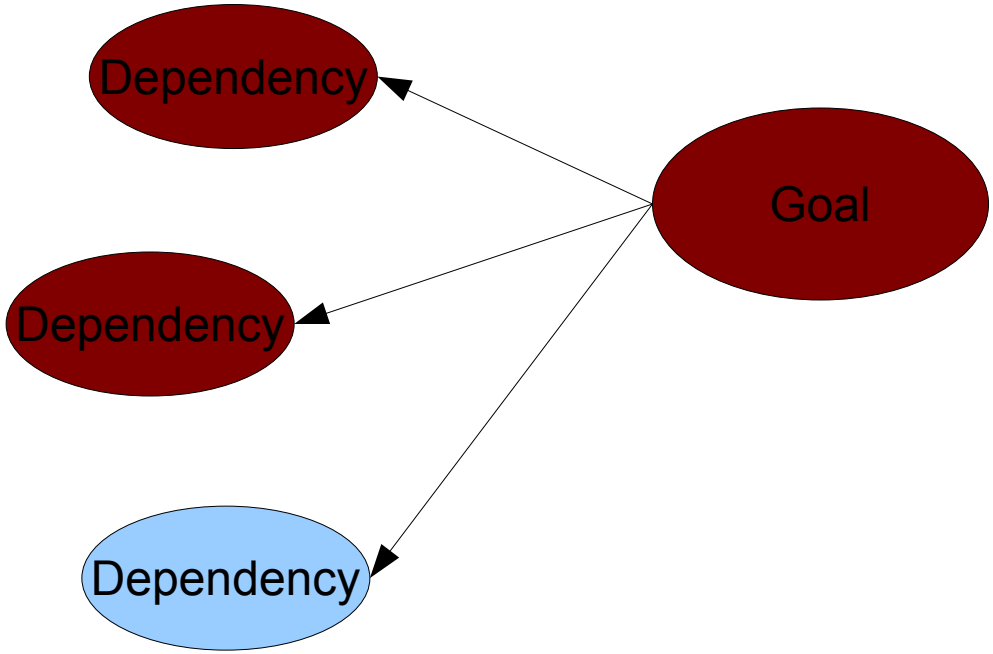
A structured refactoring method

Fix each of the dependencies with the simplest solution that could work



A structured refactoring method

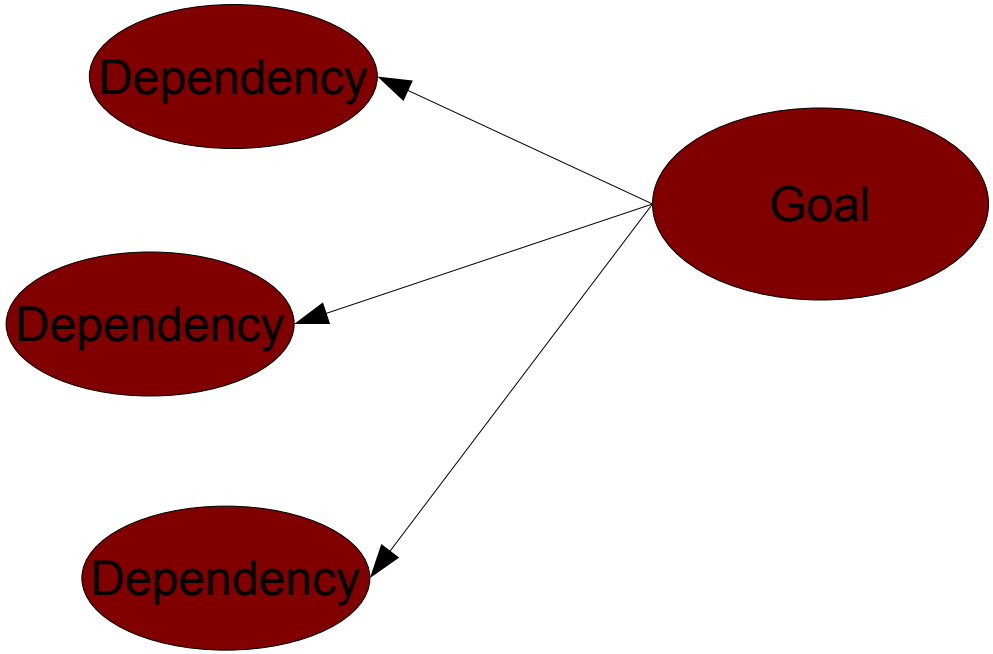
Fix each of the dependencies with the simplest solution that could work



SIGMA

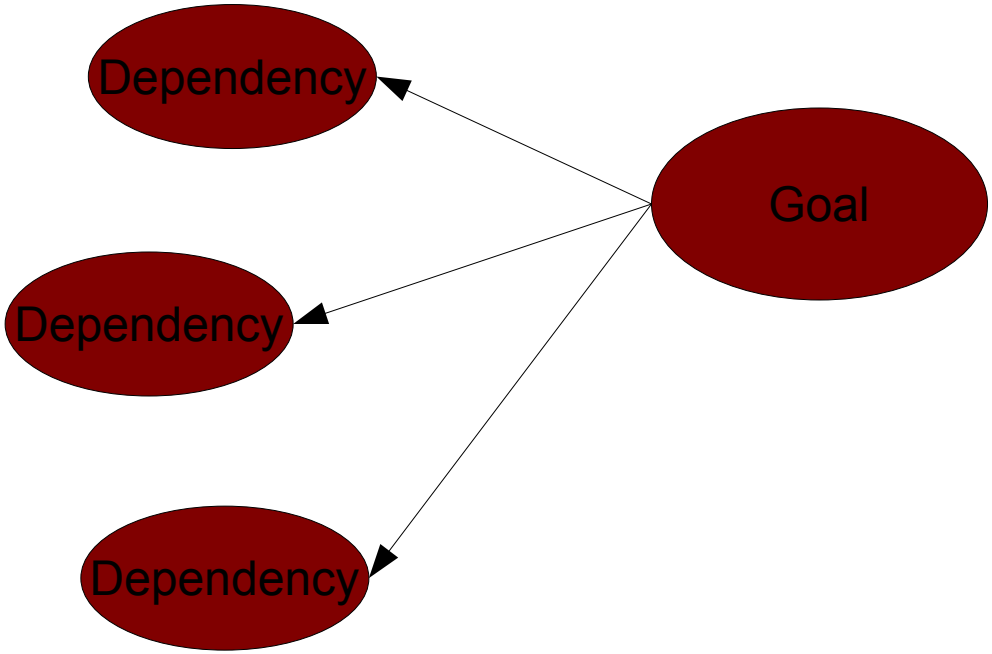
A structured refactoring method

Fix each of the dependencies with the simplest solution that could work



A structured refactoring method

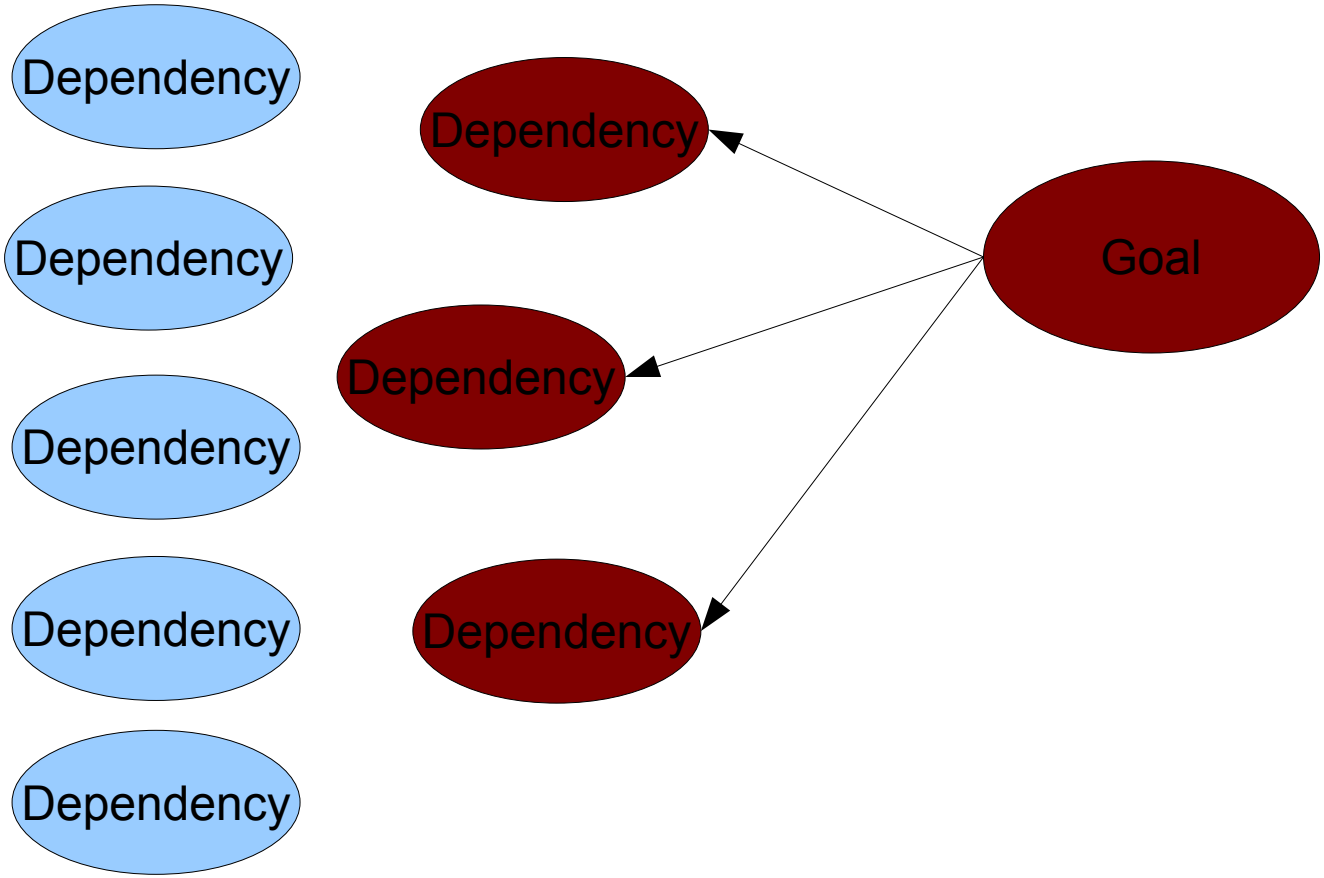
Run all tests again



SIGMA

A structured refactoring method

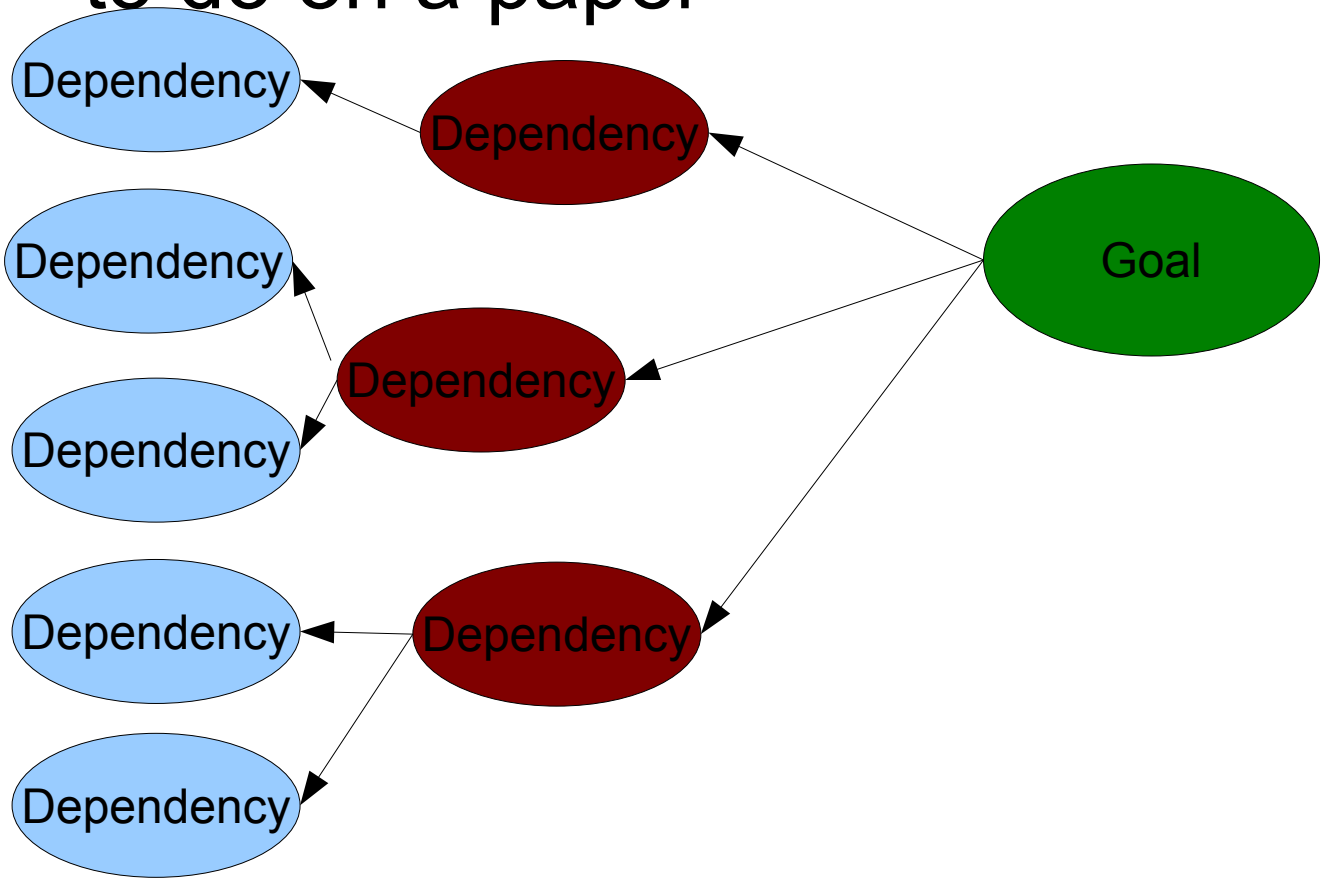
We found more dependencies



SIGMA

A structured refactoring method

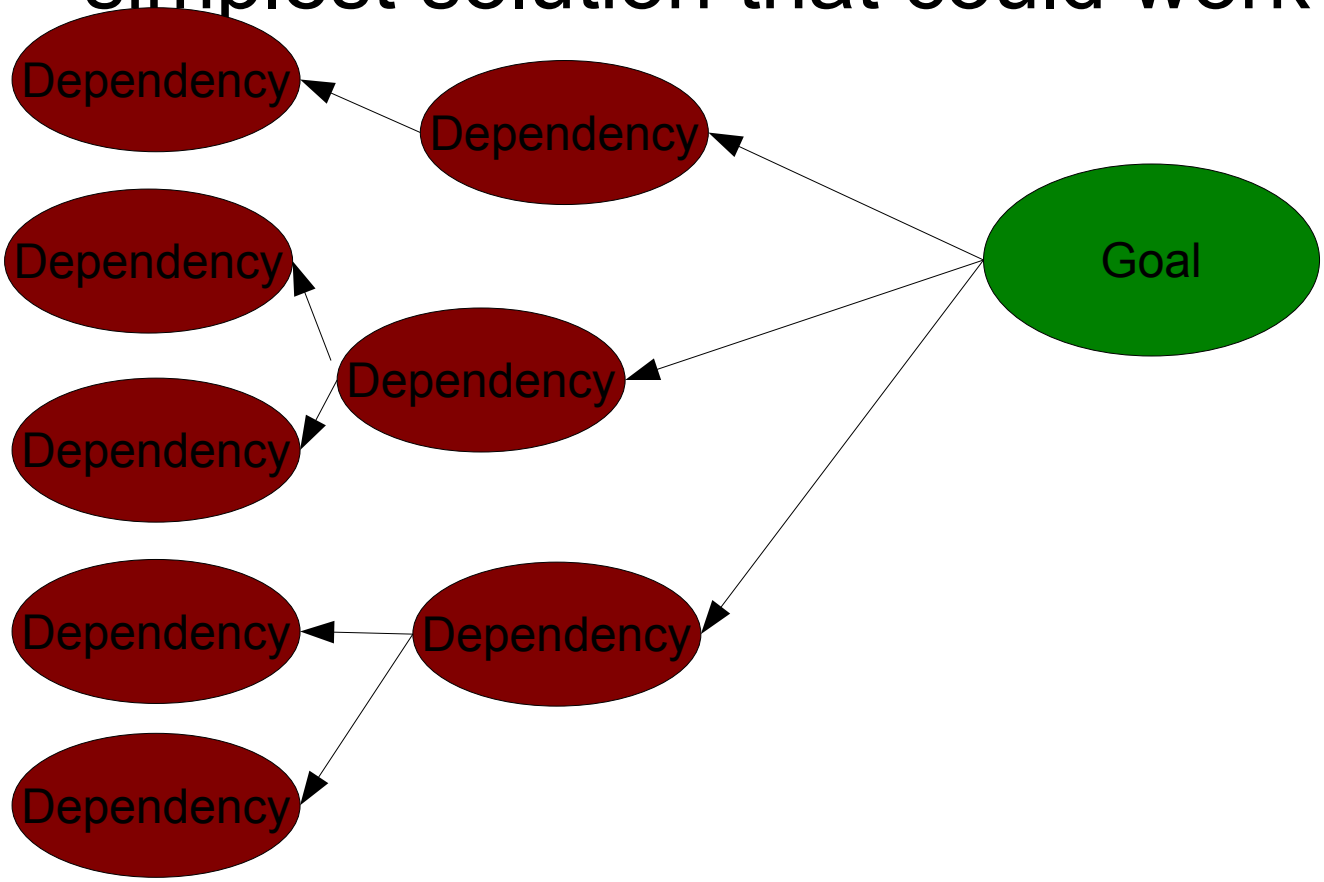
Rollback the goal and note what you wanted to do on a paper



SIGMA

A structured refactoring method

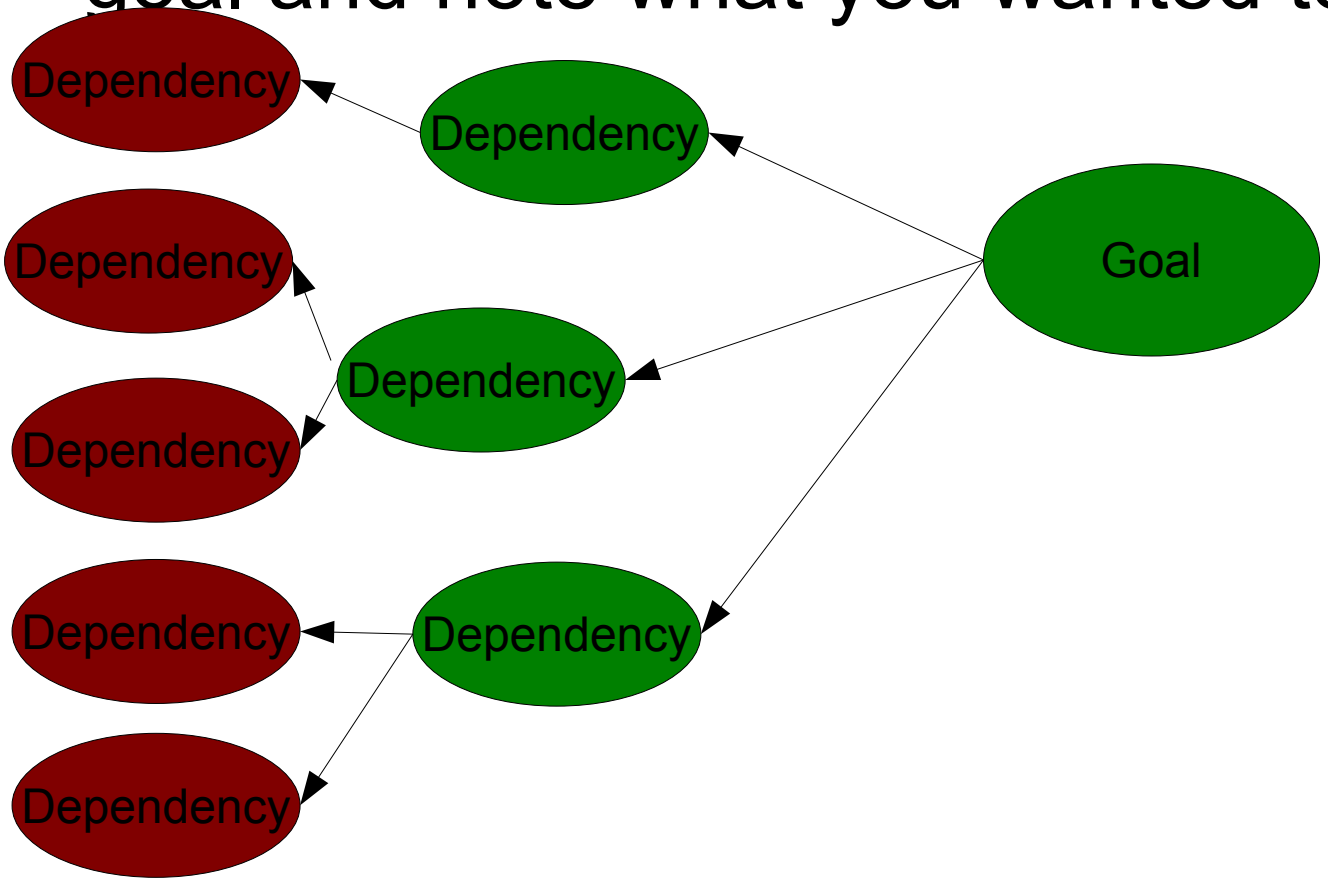
Fix each of the dependencies with the simplest solution that could work



SIGMA

A structured refactoring method

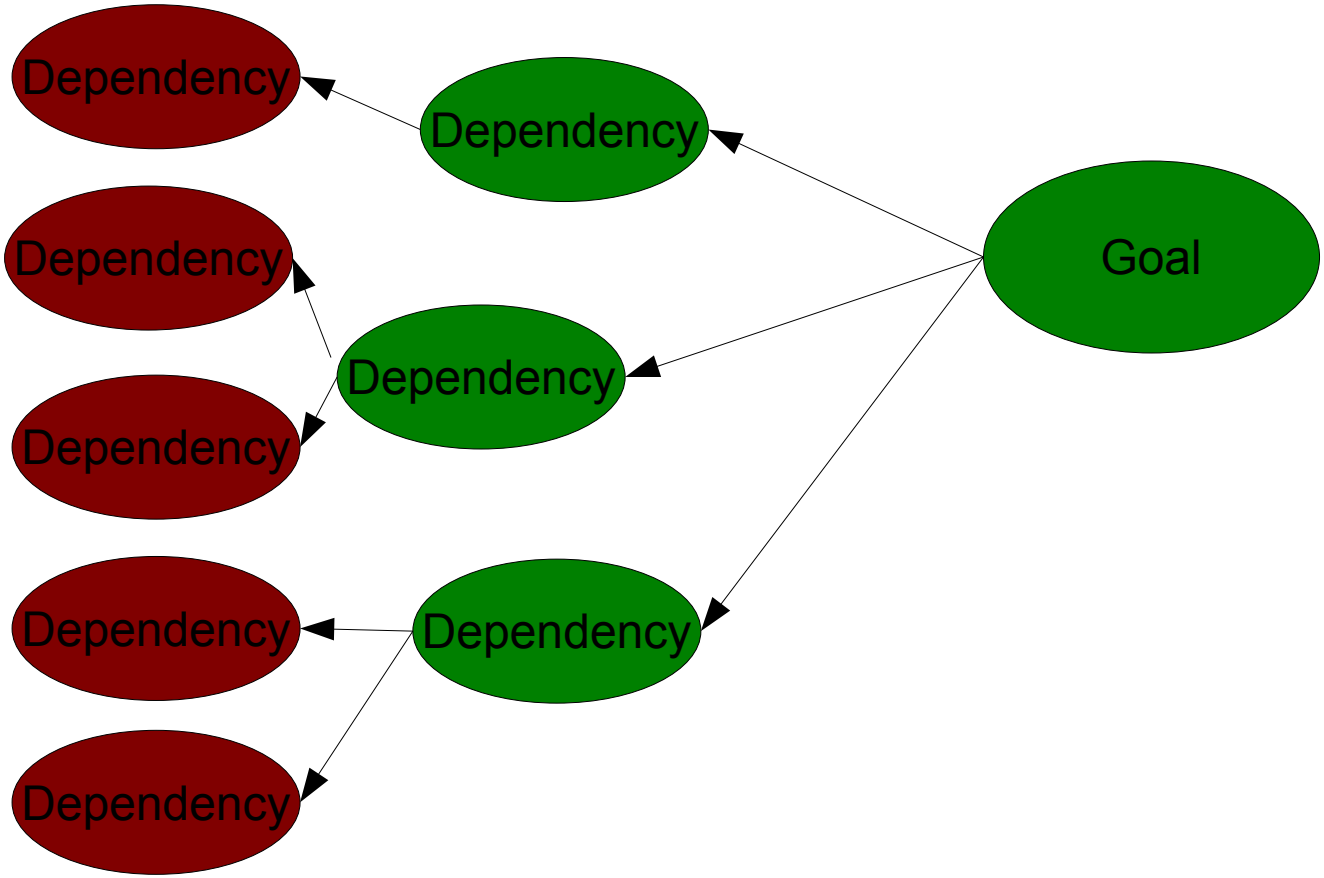
Rollback the dependencies closets to the goal and note what you wanted to do



SIGMA

A structured refactoring method

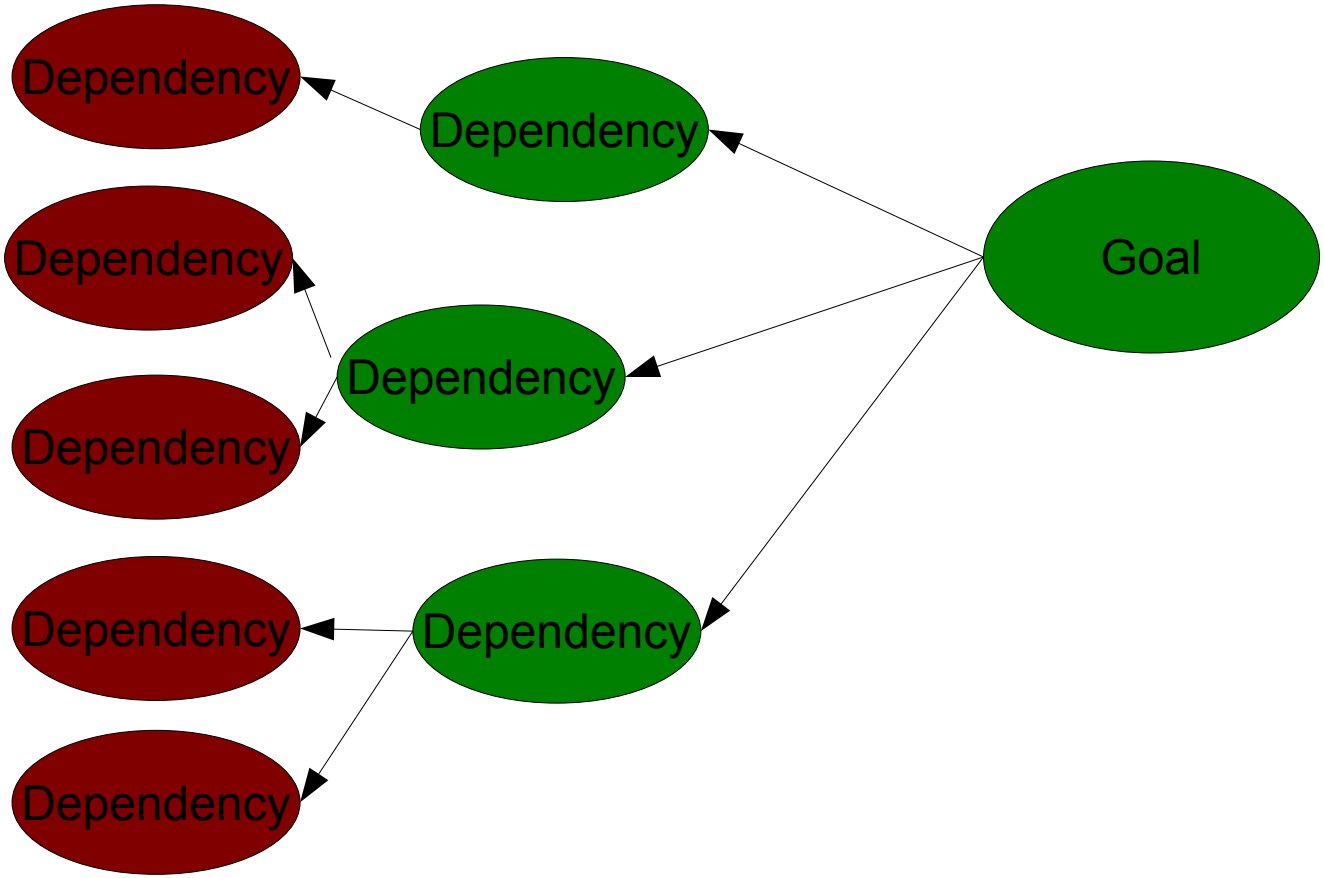
Run all tests again



SIGMA

A structured refactoring method

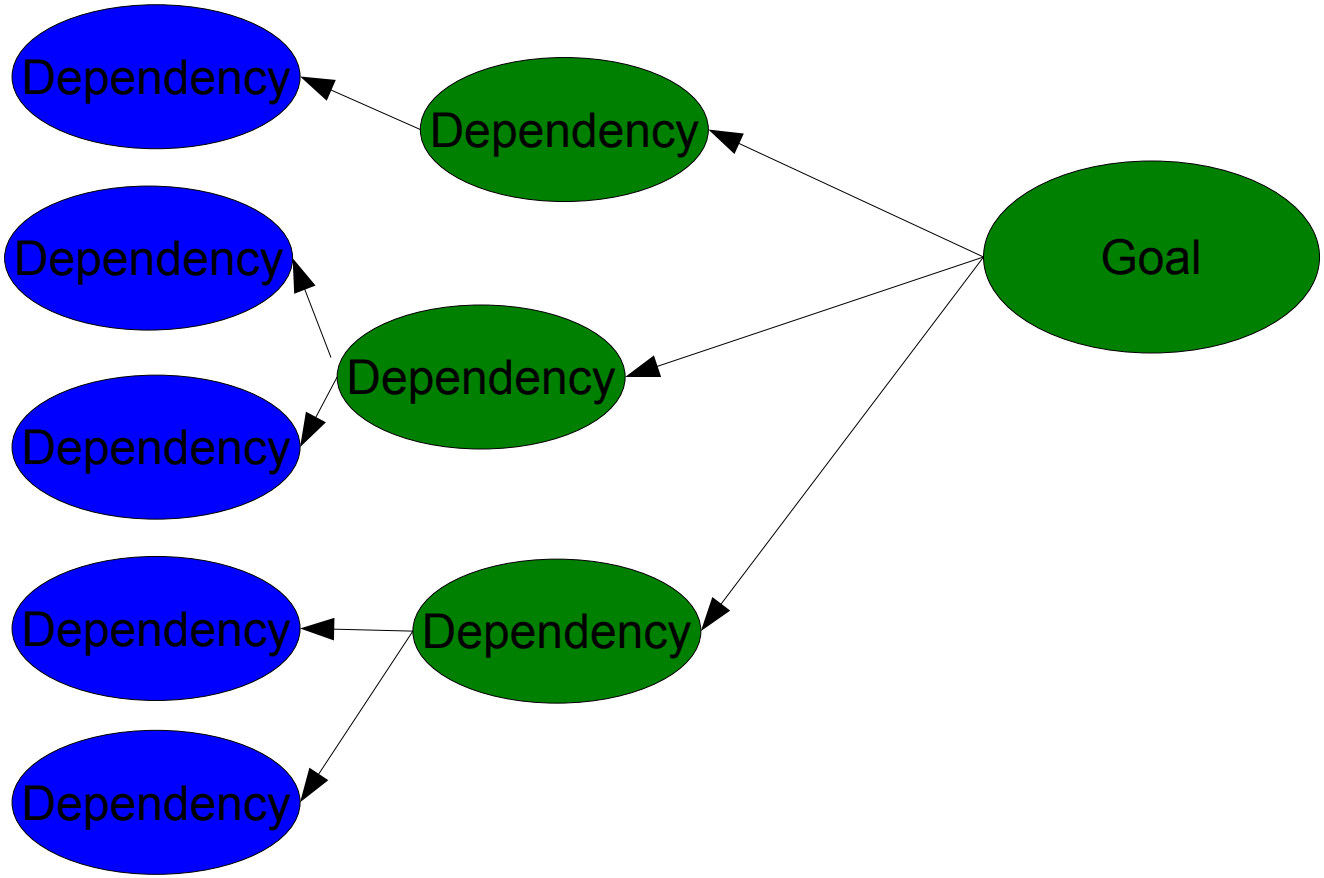
We didn't find any more dependencies



SIGMA

A structured refactoring method

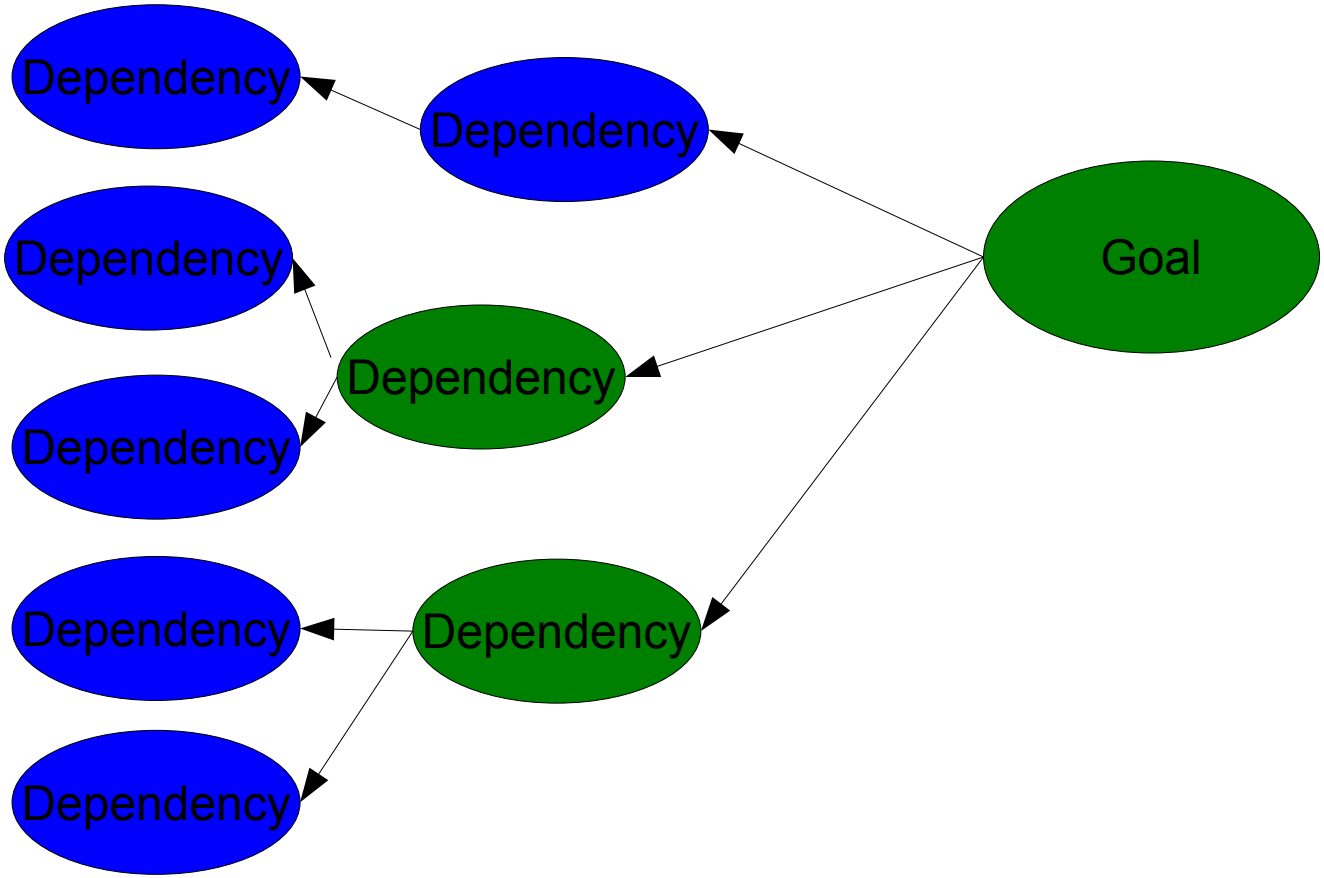
Checkin the fixed dependencies



SIGMA

A structured refactoring method

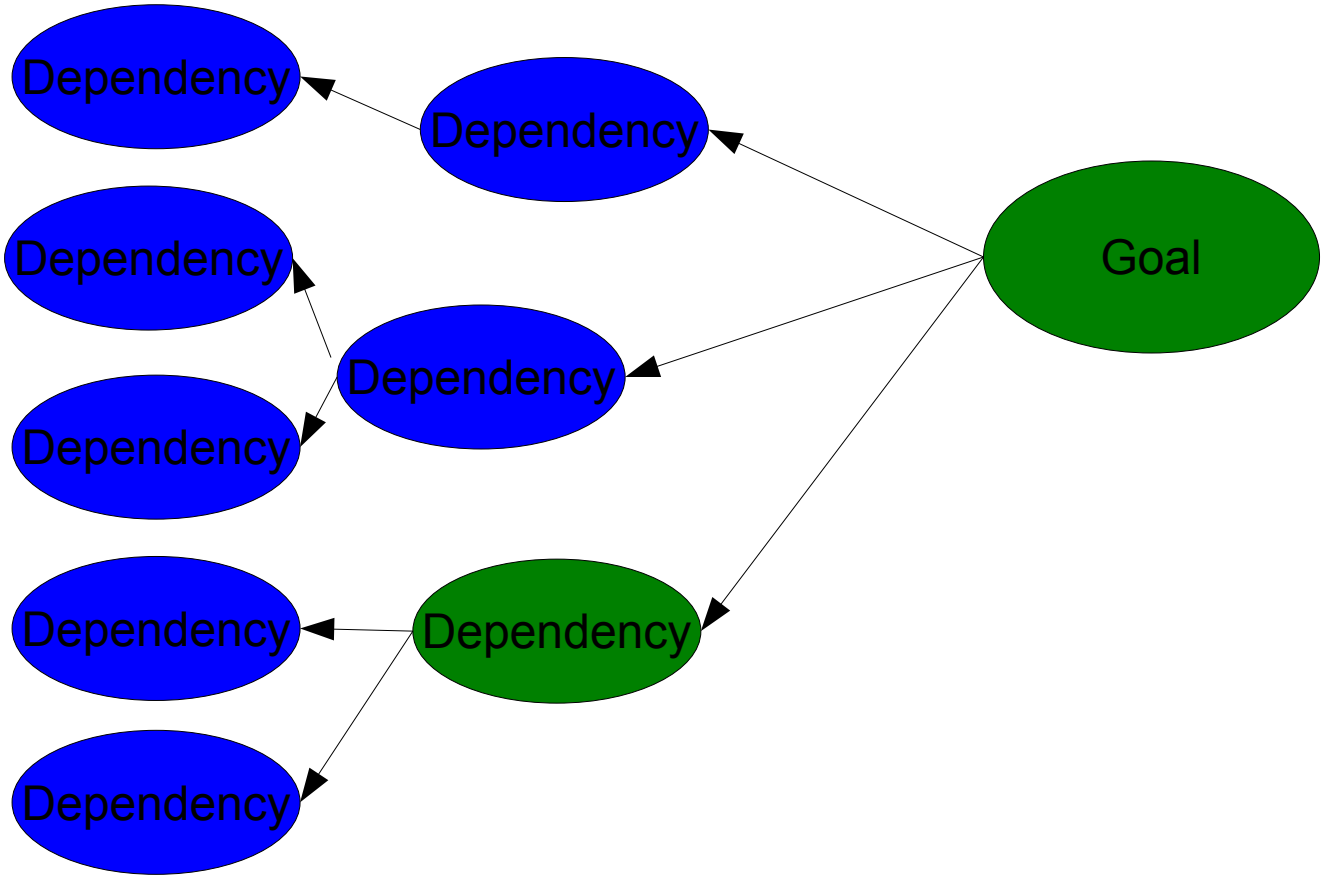
Fix and checkin each of the dependencies



SIGMA

A structured refactoring method

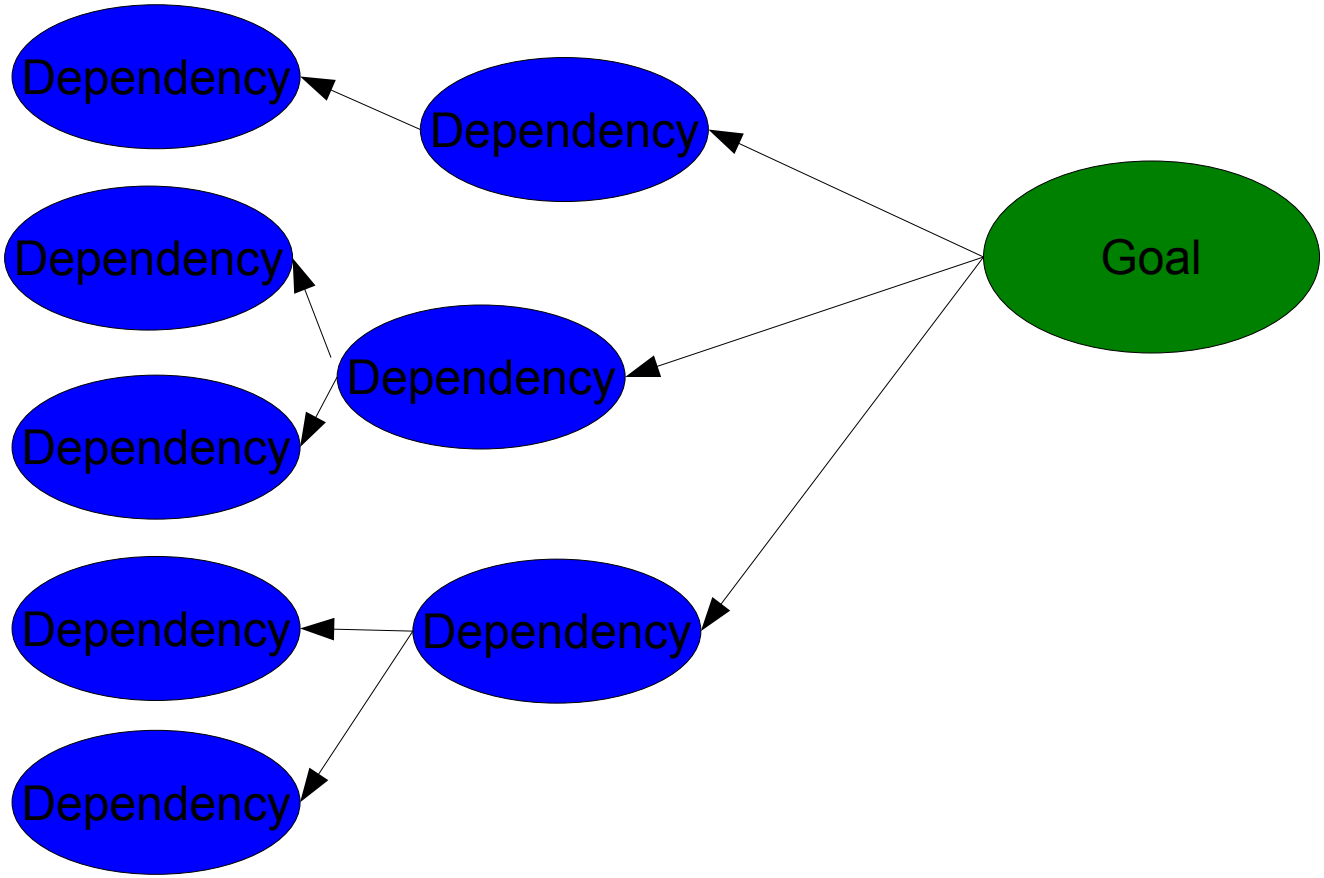
Fix and checkin each of the dependencies



SIGMA

A structured refactoring method

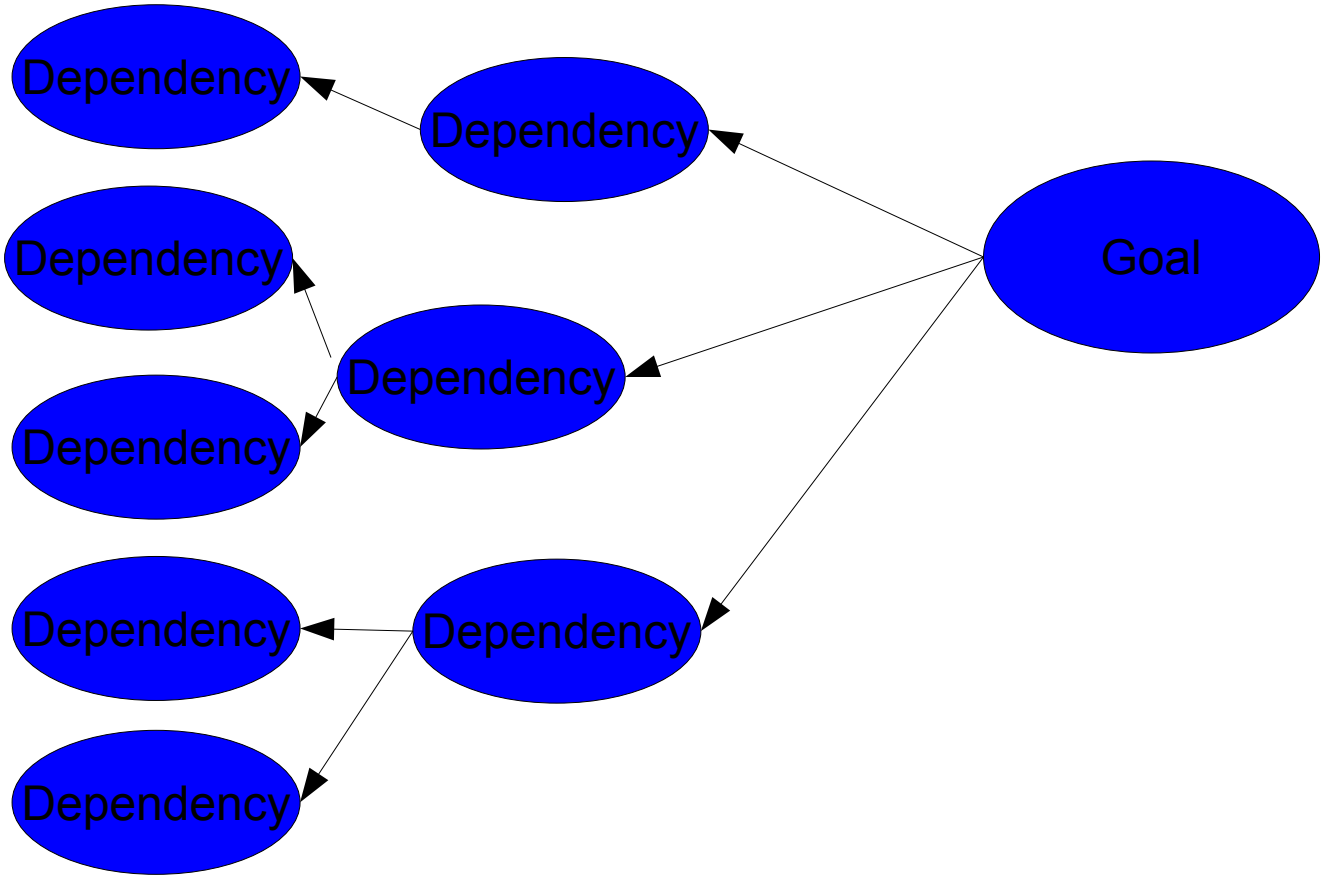
Fix and checkin each of the dependencies



SIGMA

A structured refactoring method

Finally fix and checkin the goal



SIGMA

A structured refactoring method

The most important steps are

- Run all tests
- Fix everything with the simplest possible solution
- Rollback from the origin
  - First the goal
  - Then the nodes next to the goal
- Only checkin leafs
- Checkin is only allowed when all tests passes
- Fix one leaf at a time
- Always keep the system in a working state

A structured refactoring method

Some times called the mikado method

SIGMA

The way from technical debt

Change the working environment

- Technical problems are symptoms of process problems

Education

- Always strive for more knowledge

Code improvements

- Refactor relentlessly

Add technical debt issues as tasks

- Visualise your debt

Automate testing

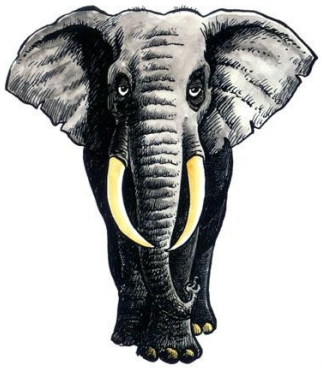
- Have computers test everything as often as possible

SIGMA

Conclusion

Paying a technical debt

- The same way as you eat an elephant
- One piece at a time
- Don't rush
- Allow it to take some time
- Avoid adding technical debt while paying



Technical debt is

Bad code that slows you down

A loan of time from the future

SIGMA

Disposition

Introduction

Symptoms

Reasons and solutions

A method for refactoring

Resources

SIGMA

Resources

Ward Cunningham

- OOPSLA '92 - http://c2.com/doc/oopsla92.html
- http://www.c2.com/cgi/wiki?TechnicalDebt

Martin Fowler

- http://www.martinfowler.com/
- Refactoring, ISBN 0201485672

Robert C Martin – Clean Code, ISBN 0132350882

Michael C. Feathers – Working effectively with legacy code, ISBN 0131177052

Steve McConnell - http://blogs.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx

SIGMA

Technical Debt - Goal

Get a metaphor that can be communicated to non technical people

Know that all reasons are not technical

Know that some reasons cannot be foreseen

Have a structured refactoring method for large re-factorings

SIGMA

Technical Debt

Thomas Sundberg

Consultant, Developer
Stockholm, Sweden
Sigma Solutions AB

thomas.sundberg@sigma.se
@thomassundberg
http://thomassundberg.wordpress.com

SIGMA