MGL7460 Réalisation et maintenance de logiciels

Guy Tremblay
Professeur

Département d'informatique
UQAM
http://www.labunix.uqam.ca/~tremblay

8 septembre 2016

Description officielle

Description «officielle» du cours MGL7460

Rôle de la réalisation et de la maintenance dans le cycle de vie du logiciel. Évolution et maintenance du logiciel. Méthodes propres à étendre la durée de vie. Sélection de la méthode appropriée de réalisation. Prototypage. Mise au point. Gestion de la maintenance. Réutilisation et rétro-ingénierie des logiciels. L'interaction entre réalisation et maintenance sera traitée tout au long du cours.

Background du cours

Question :

À quoi correspondent les notions de «Réalisation et maintenance» ?

Une ébauche possible de réponse dans deux références intéressantes

- Document Web : P. Bourque and R.E. Fairley, editors. Guide to the Software Engineering Body of Knowledge (Version 3.0). IEEE Computer Society, 2014.
  `http://www.computer.org/web/swebok/v3`

- Volume suggéré (COOP) : A. Hunt and D. Thomas. The Pragmatic Programmer—From Journeyman to Master. Addison-Wesley, 2000.

Et de nombreuses autres références :
`http://www.labunix.uqam.ca/~tremblay/MGL7460/Liens/references.pdf`

Guide to the SWEBOK

SWEBOK® V3.0

Guide to the Software Engineering Body of Knowledge

SWEBOK® V3.0

Guide to the Software Engineering Body of Knowledge

Editors

Pierre Bourque
Richard E. (Dick) Fairley

IEEE computer society

Objectifs du *Guide to the SWEBOK*

- Définir ce qu'est le génie logiciel en définissant un **guide** vers le corpus des connaissances en génie logiciel

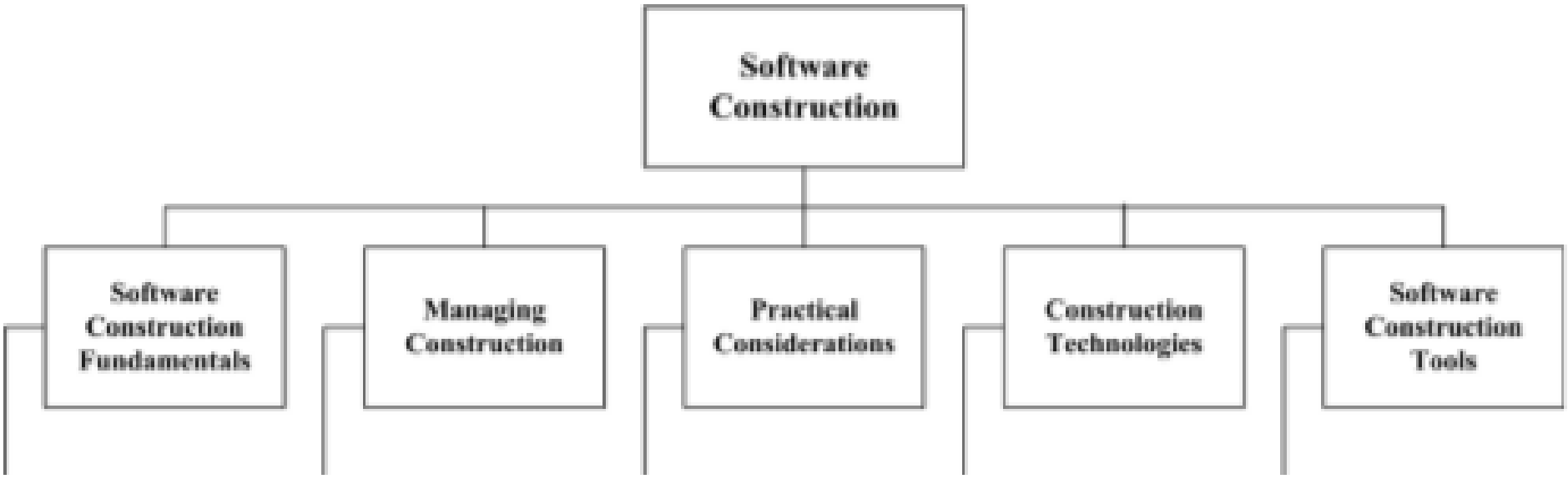
Objectifs du *Guide to the SWEBOK* (bis)

1. *To promote a consistent view of software engineering worldwide*
2. *To specify the scope of, and clarify the place of software engineering with respect to other disciplines such as computer science, project management, computer engineering, and mathematics*
3. *To characterize the contents of the software engineering discipline*
4. *To provide a topical access to the Software Engineering Body of Knowledge*
5. *To provide a foundation for curriculum development and for individual certification and licensing material*

Les différents *Knowledge Areas (KA)*

Table I.1. The 15 SWEBOK KAs
Software Requirements
Software Design
Software Construction
Software Testing
Software Maintenance
Software Configuration Management
Software Engineering Management
Software Engineering Process
Software Engineering Models and Methods
Software Quality
Software Engineering Professional Practice
Software Engineering Economics
Computing Foundations
Mathematical Foundations
Engineering Foundations

Software Construction Knowledge Area



Contenu plus détaillé du *Software Construction KA* : Software Construction Fundamentals

- Minimizing complexity
- Anticipating change
- Constructing for verification
- Standards in construction

Contenu plus détaillé du *Software Construction KA* : Managing Construction

- Construction in life cycle models
- Construction planning
- Constructing measurement

Contenu plus détaillé du *Software Construction KA* : Practical Considerations

- Construction design
- Construction languages
- Coding
- Construction testing
- Construction for reuse
- Construction with reuse
- Construction quality
- Integration

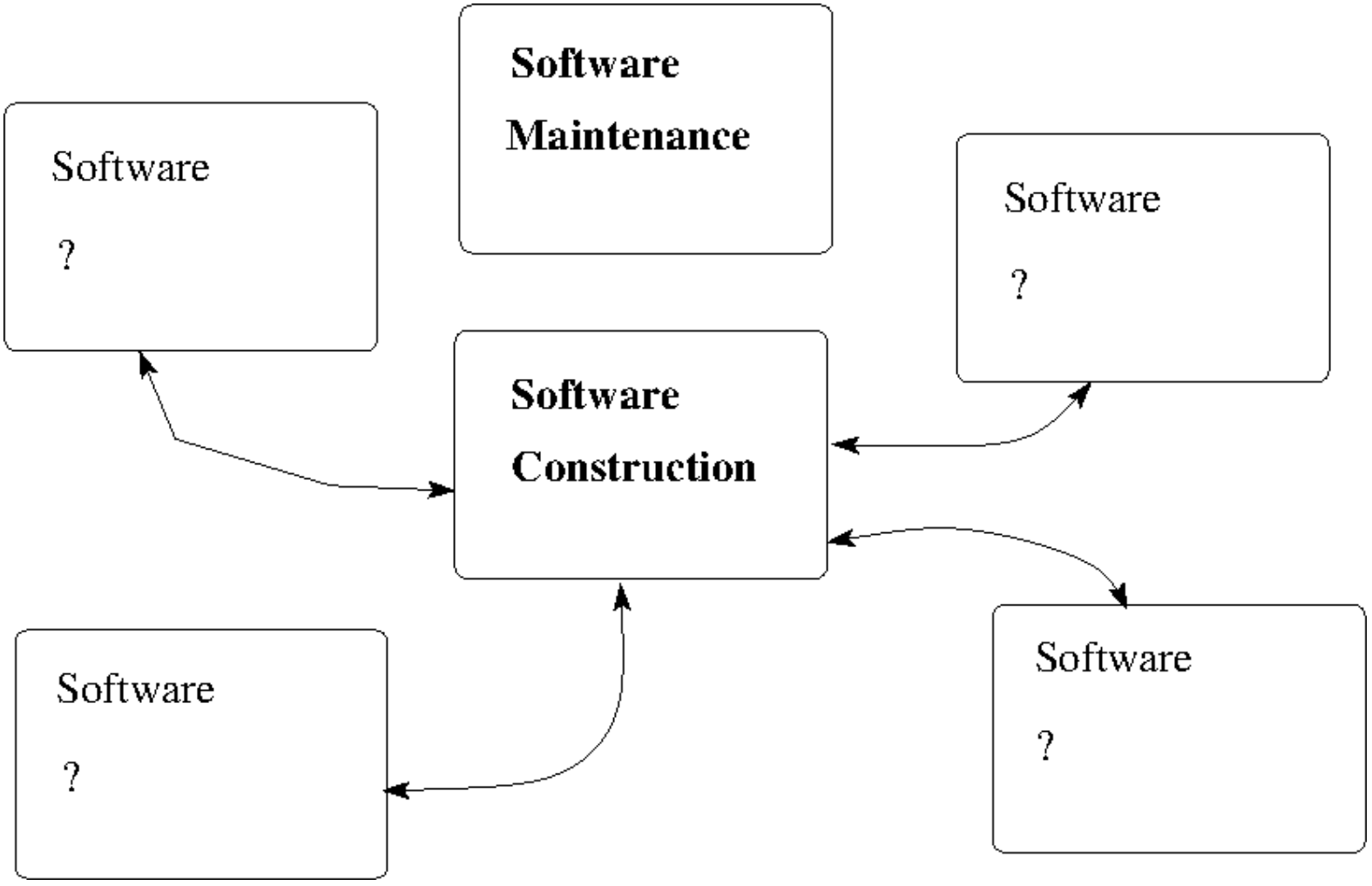
Contenu plus détaillé du *Software Construction KA* : Construction Technologies

- API design and use
- Object-oriented runtime issues
- Parametrization and generics
- Assertions, design by contract, and defensive prog.
- Error handling, exception handling, and fault tolerance
- Executable models
- State-based and table-driven construction techniques
- Runtime configuration and internationalization
- Grammar-based input processing
- Concurrency primitives
- Middleware
- Distributed software
- Heterogeneous systems
- Performance analysis and tuning
- Platform standards
- Test-first programming

Contenu plus détaillé du *Software Construction KA* : Software Construction Tools

- Development environments
- GUI builders
- Unit testing tools
- Profiling, performance analysis, and slicing tools

Les KAs reliés à *Software Construction* ?



Les KAs reliés à *Software Construction* ?

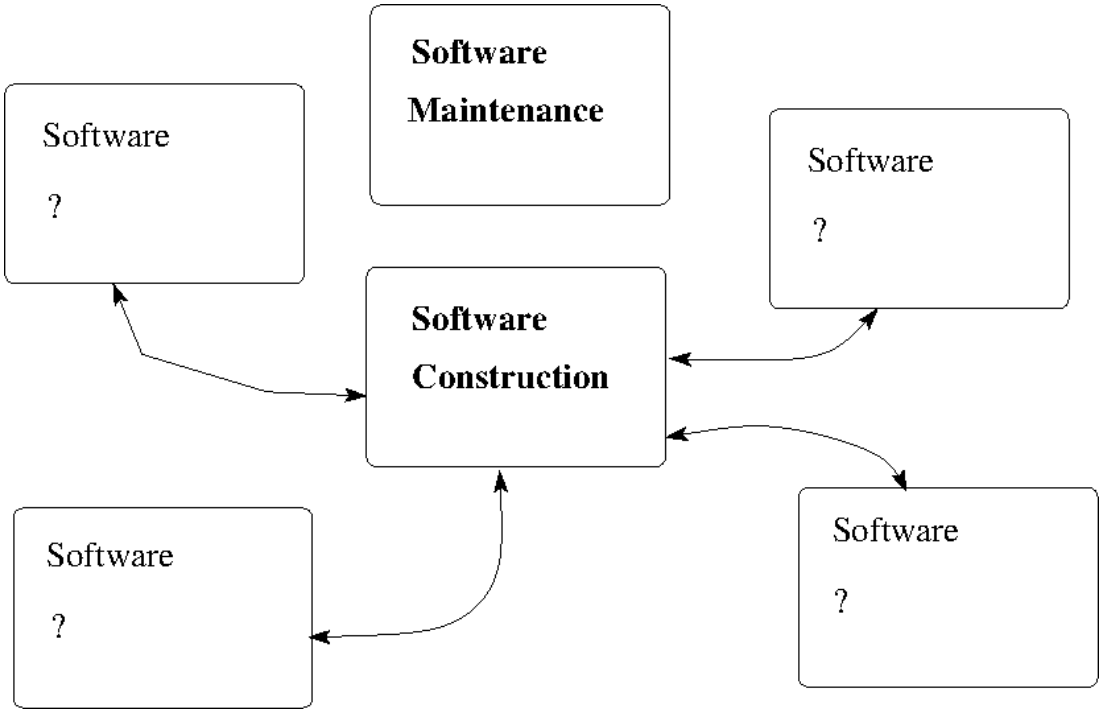
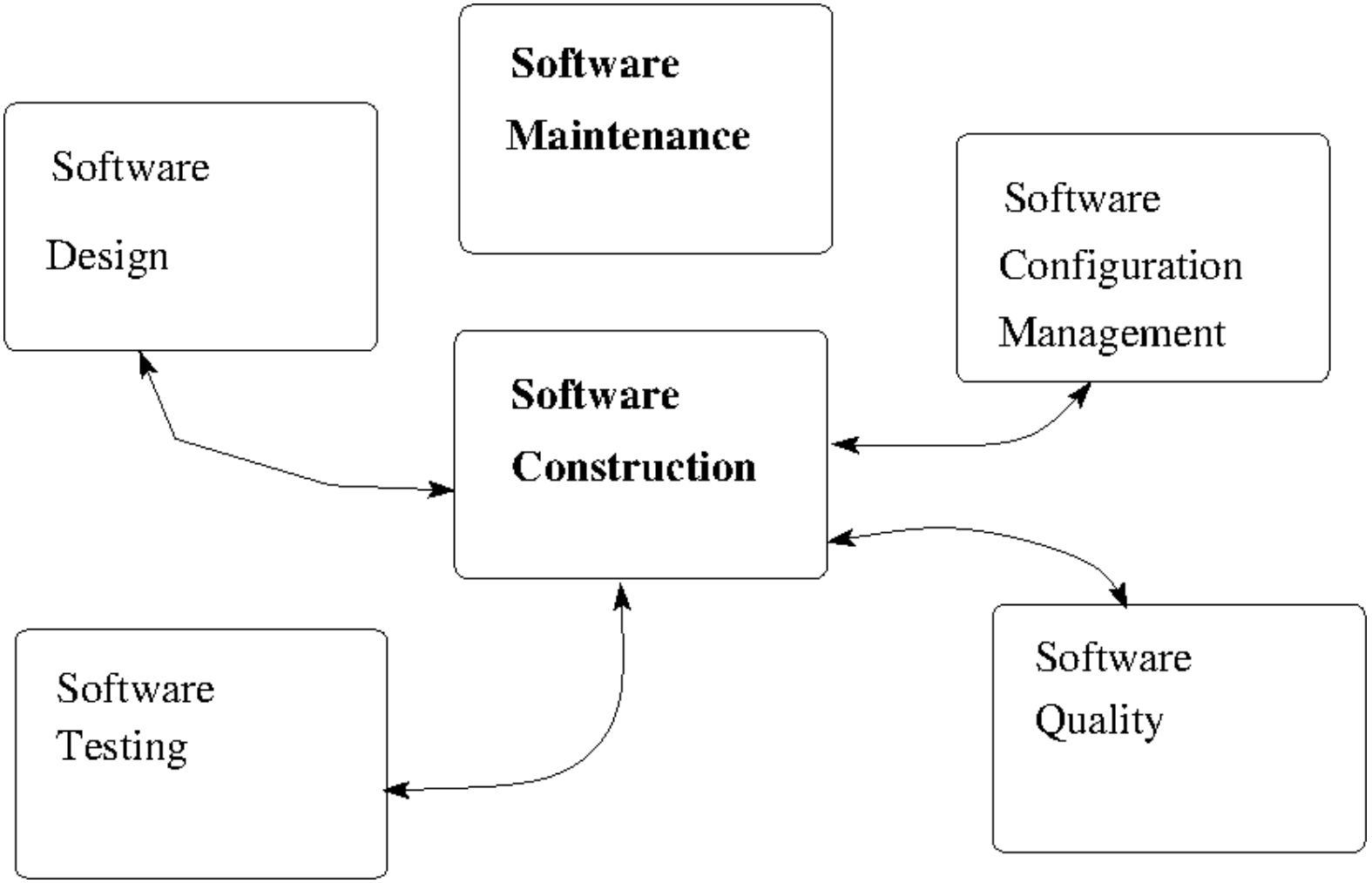
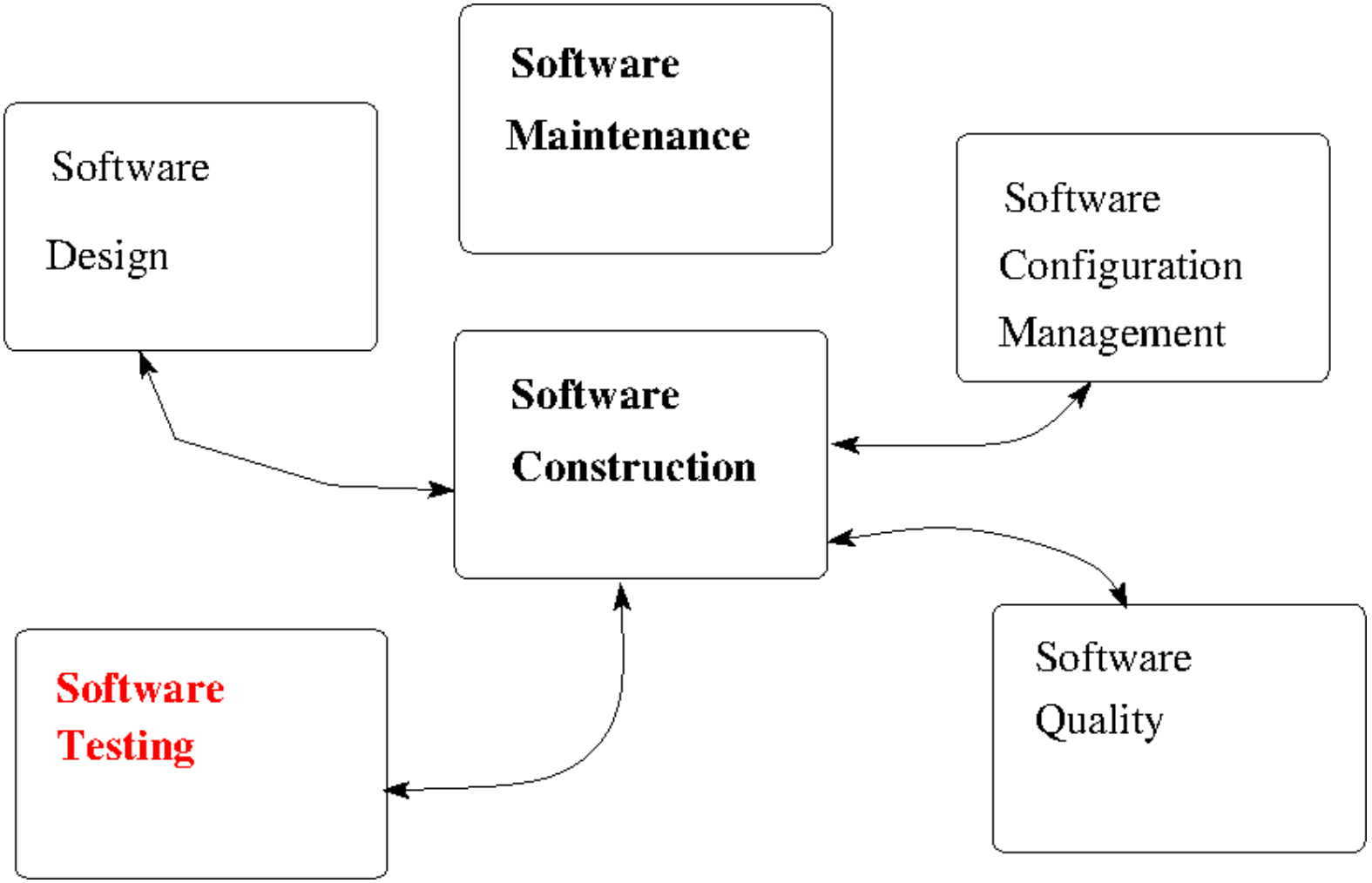


Table I.1. The 15 SWEBOK KAs
Software Requirements
Software Design
Software Construction
Software Testing
Software Maintenance
Software Configuration Management
Software Engineering Management
Software Engineering Process
Software Engineering Models and Methods
Software Quality
Software Engineering Professional Practice
Software Engineering Economics
Computing Foundations
Mathematical Foundations
Engineering Foundations

Les KAs reliés à *Software Construction* !

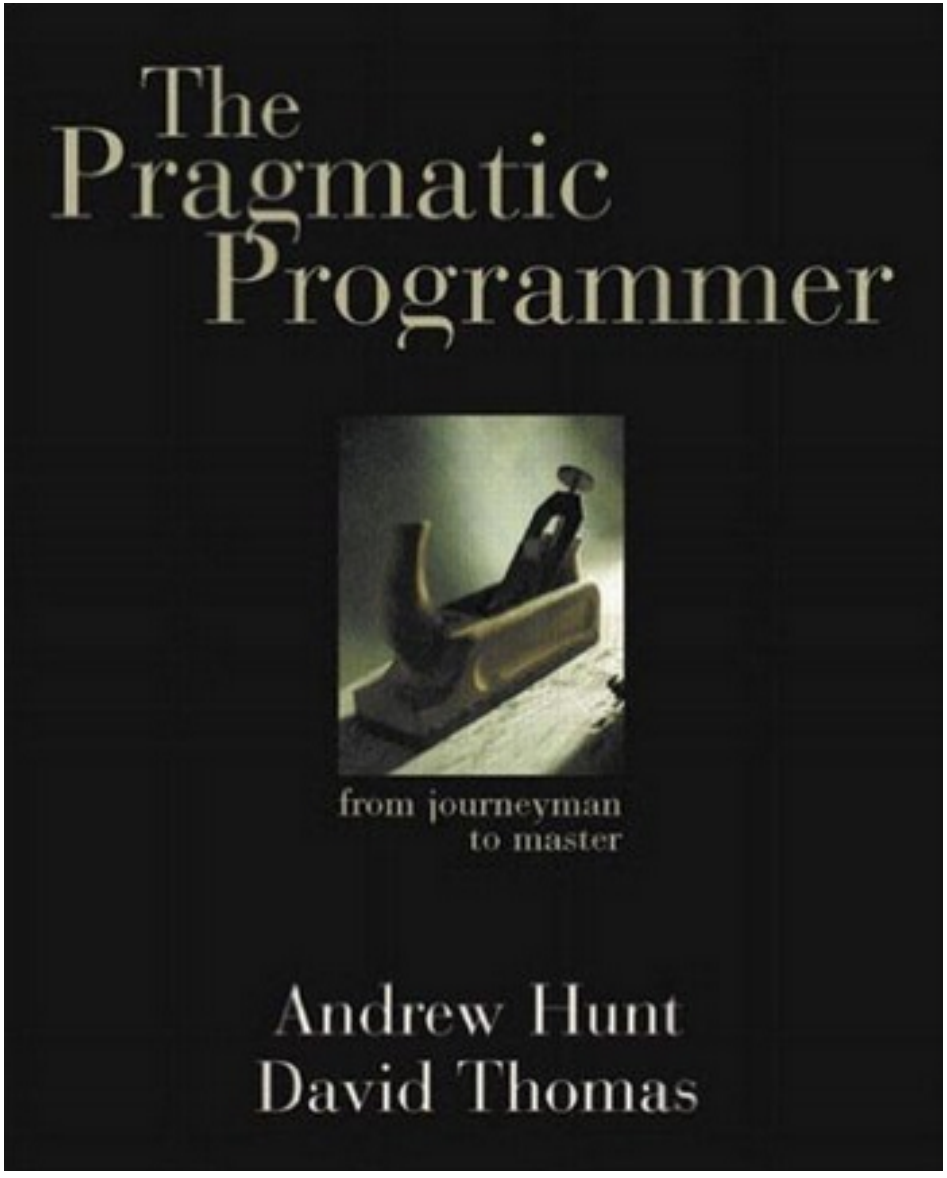


Les KAs reliés à *Software Construction* !



The Pragmatic Programmer—From Journeyman to Master

« *The Pragmatic Programmer* » est un livre paru en 2000



Ses auteurs : Andrew Hunt et Dave Thomas

Andrew Hunt et Dave Thomas sont deux des signataires du «Manifeste Agile»

Vous connaissez la «Programmation Agile» ?

Hunt & Thomas ont fondé la maison d'édition The Pragmatic Bookshelf : plus de 280 titres parus (2015) !

Read [the] book "The Pragmatic Programmer", and you'll learn how to :

- Fight software rot.
- Catalyze change.
- Avoid the trap of duplicating knowledge.
- Write flexible, dynamic and adaptable code.
- Harness the power of basic tools.
- Avoid programming by coincidence.
- Bullet-proof your code with contracts, assertions and exceptions.
- Capture real requirements.
- Keep formal tools in their place.
- Test ruthlessly and effectively.
- Delight your users.
- Build teams of pragmatic programmers.
- Take responsibility for your work and career.
- Make your developments more precise with automation.

Questions. . .

- Qu’est-ce que le «*software rot*» ?

Questions. . .

- Qu’est-ce que le «*software rot*» ?

- Qu’est-ce qu’un contrat ?

Questions. . .

- Qu’est-ce que le «*software rot*» ?
- Qu’est-ce qu’un contrat ?
- Comment peut-on tester «*ruthlessly and effectively*» ?

Questions. . .

- Qu'est-ce que le «*software rot*» ?
- Qu'est-ce qu'un contrat ?
- Comment peut-on tester «*ruthlessly and effectively*» ?
- À quoi réfère l'idée d'*automatisation* dans «*Make your developments more precise with automation*» ?

Les auteurs introduisent divers *tips*

Tips : Caractéristiques, pratiques, approches partagées par les «programmeurs pragmatiques», exprimées sous forme de conseils

Quelques *tips* généraux

- *Care About Your Craft*
  Why spend your life developing software unless you care about doing it well ?

- *Invest Regularly in Your Knowledge Portfolio*
  Make learning a habit.

Quelques *tips* en lien avec l'écriture de code

- *Always Use Source Code Control*
  Source code control is a time machine for your work—you can go back.
- *DRY—Don't Repeat Yourself*
  Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.
- *Use the Power of Command Shells*
  Use the shell when graphical user interfaces don't cut it.
- *Use Exceptions for Exceptional Problems*
  Exceptions can suffer from all the readability and maintainability problems of classic spaghetti code. Reserve exceptions for exceptional things.

Questions. . .

- Quel est la cause la plus courante de code «non DRY» ?

Questions. . .

- Quel est la cause la plus courante de code «non DRY» ?

- Combien d'entre vous utilisez régulièrement un *shell* Unix/Linux ?

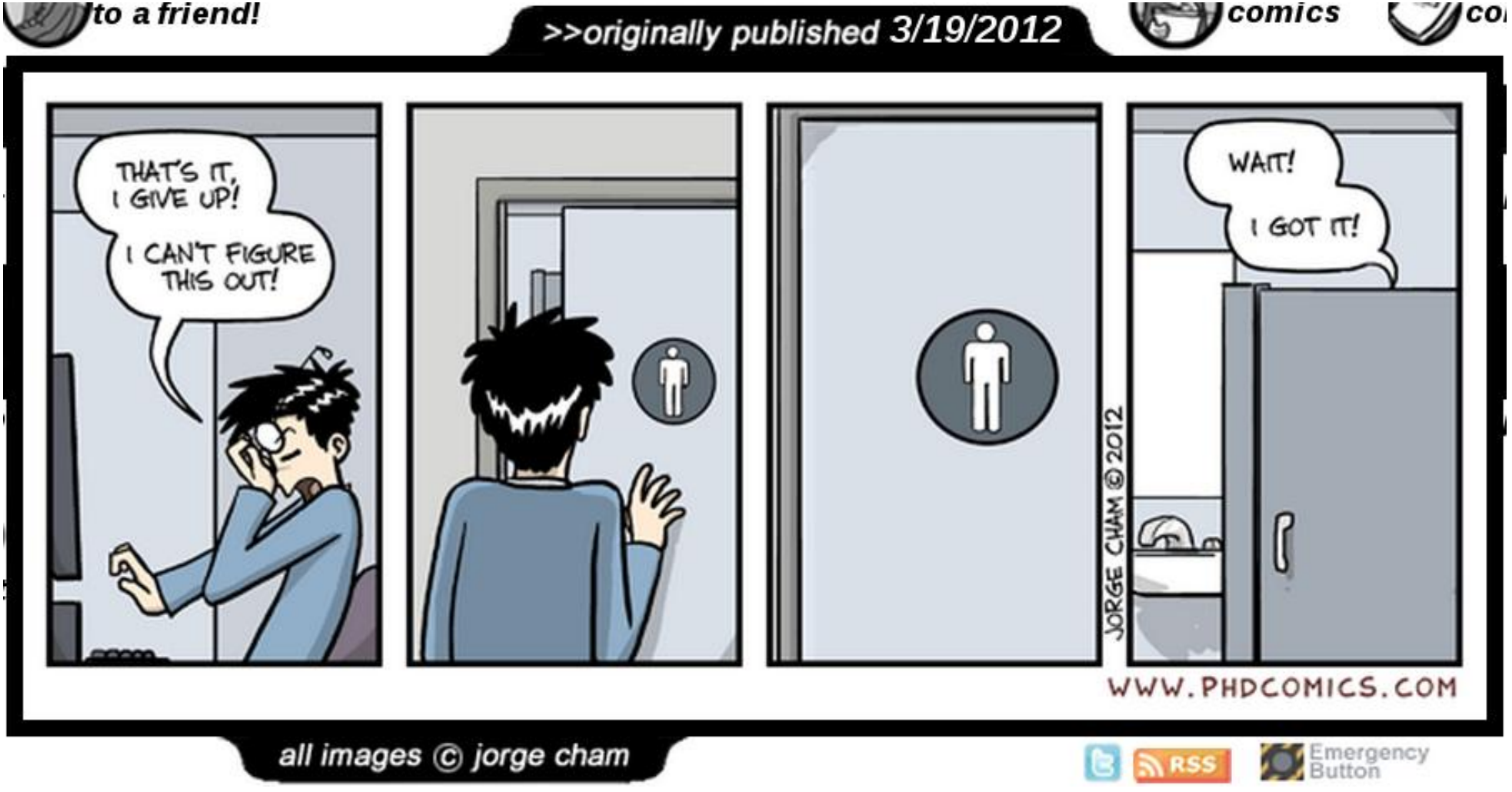
Quelques *tips* en lien avec le débogage

- *Don't Panic When Debugging*
  Take a deep breath and THINK ! about what could be causing the bug.

- *«select» Isn't Broken*
  It is rare to find a bug in the OS or the compiler, or even a third-party product or library. The bug is most likely in the application.

- *Find Bugs Once*
  Once a human tester finds a bug, it should be the last time a human tester finds that bug. Automatic tests should check for it from then on.

Débogage et autres problèmes : Step away from the keyboard to solve hard problems.



Source : `http://phdcomics.com/comics.php`

Quelques *tips* en lien avec les tests

- *Test Your Software, or Your Users Will*
  Test ruthlessly. Don't make your users find bugs for you.

- *Coding Ain't Done 'Til All the Tests Run*
  'Nuff said.

- *Test Early. Test Often. Test Automatically.*
  Tests that run with every build are much more effective than test plans that sit on a shelf.

- *Refactor Early, Refactor Often*
  Just as you might weed and rearrange a garden, rewrite, rework, and re-architect code when it needs it. Fix the root of the problem.

Quelques *tips* en lien avec les contrats

- *Design with Contracts*
  Use contracts to document and verify that code does no more and no less than it claims to do.

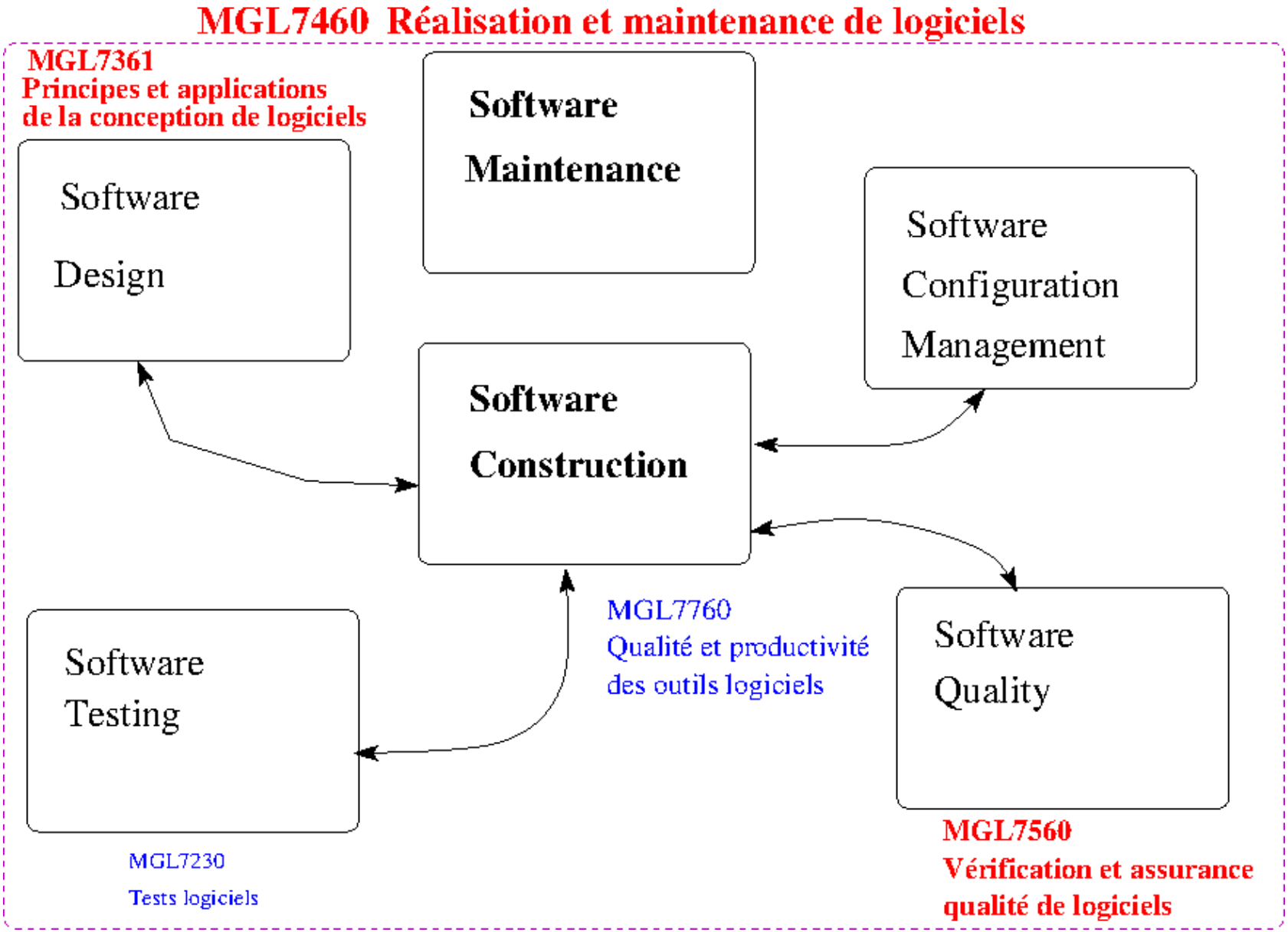
- *Use Assertions to Prevent the Impossible*
  Assertions validate your assumptions. Use them to protect your code from an uncertain world.

- *Crash Early*
  A dead program normally does a lot less damage than a crippled one.

Objectifs et contenu du cours MGL7460

Portée du cours MGL7460

Légende : En rouge = **Cours obligatoire** ; En bleu = **Cours optionnel**



Et la maintenance dans tout cela ?

Most people assume that maintenance begins when an application is released, that maintenance means fixing bugs and enhancing features. We think these people are wrong. Programmers are constantly in maintenance mode. Our understanding changes day by day. New requirements arrive as we're designing or coding. Perhaps the environment changes. Whatever the reason, maintenance is not a discrete activity but a routine part of the entire development process.

A. Hunt & D. Thomas, «The Pragmatic Programmer»

Objectif général du cours MGL7460

Se familiariser avec les concepts, pratiques, techniques et outils visant à développer et maintenir des logiciels de façon professionnelle.

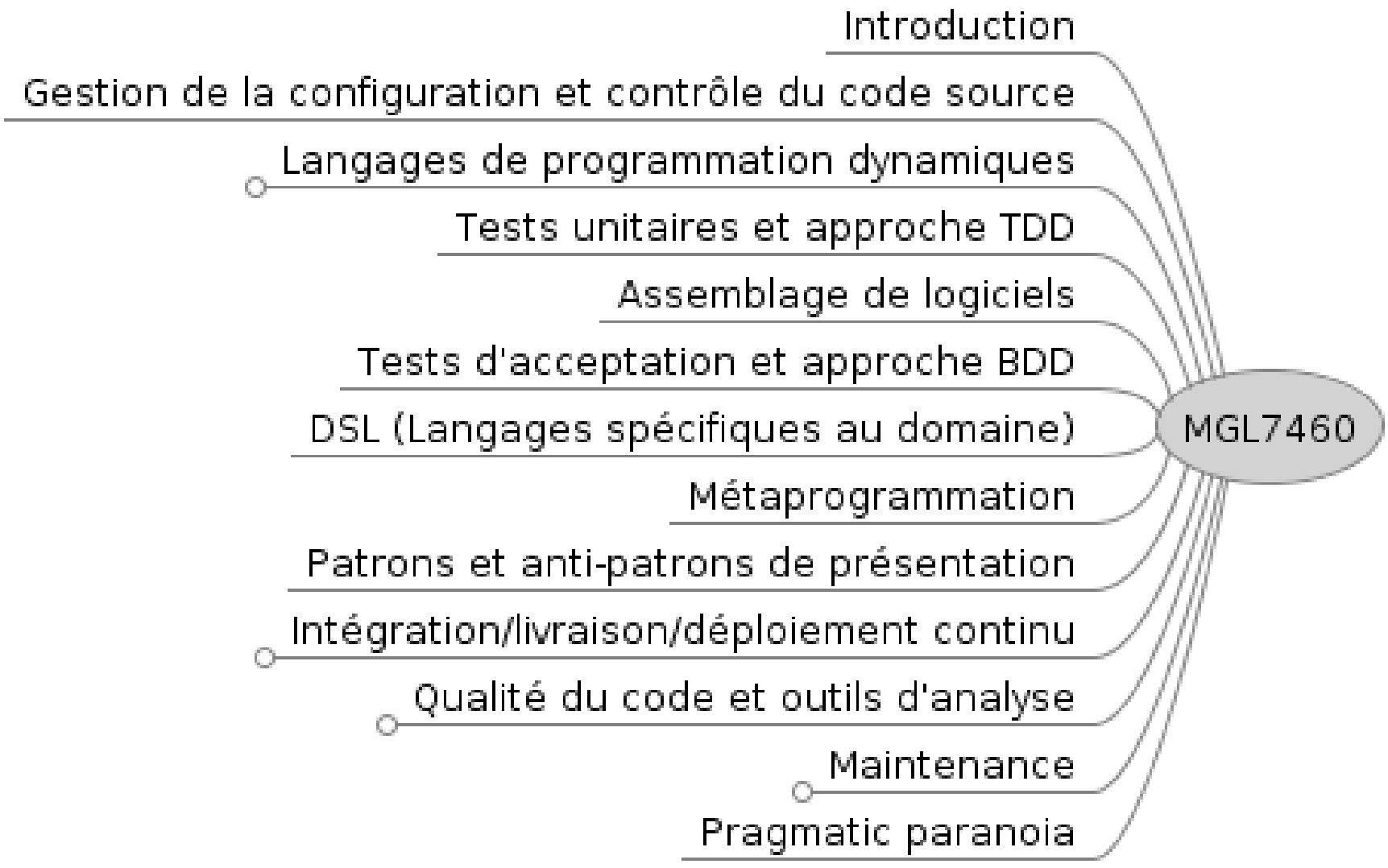
Objectifs spécifiques du cours MGL7460

R & M = Réalisation et Maintenance de logiciels

- Comprendre la problématique de la R & M.
- Situer les activités de R & M dans le cycle de vie.
- Savoir utiliser des outils de contrôle du code source.
- Comprendre l'importance des activités d'assemblage et de déploiement des logiciels.
- Connaître et utiliser des outils et techniques de test.
- Connaître et utiliser des outils d'analyse de code pour en évaluer la qualité et la maintenabilité.
- Comprendre ce qu'est la«dette technique» et le *refactoring*.
- Se familiariser avec de nouvelles approches dans le domaine et de nouveaux langages.

Contenu du cours

Basé sur mon expérience d'enseignement du cours MGL7460 à l'automne 2015



Évaluation du cours et approche pédagogique

Modalités d'évaluation

Description	**Date**	**Pondération**
Projet 1 : Ruby + Tests unitaires	13 oct.	20 %
Projet 2 + Présentation orale : Tests d'acceptation	17 & 24 nov.	25 %
Projet 3 : À déterminer	19 déc.	20 %
Participation/Rapports de participation		10 %
Examen final	15 déc.	25 %

Les projets

- En équipe d'au plus trois (3) personnes — 1, 2 ou 3
- Peut être fait seul. . . mais non recommandé — évalué comme n'importe quel autre projet
- À remettre :
  - Contrat d'équipe
  - Code source sous forme de dépôt `git` (GitHub, BitBucket)
  - Rapport écrit ($\approx$ 10–15 pages, un par équipe)
  - Rapport de participation (individuel)
- Présentation orale du deuxième projet :
  $\approx$ 15–20 minutes + questions

Le déroulement des cours

1. Première partie :
   Présentation de divers thèmes et sujets par le professeur

Le déroulement des cours

1. Première partie :
   Présentation de divers thèmes et sujets par le professeur

2. Deuxième partie :
   - Activité pratique en laboratoire (PK-4665)
   - Rencontre avec les équipes pour faire le point sur les projets (*coaching*)

Références

Manuel suggéré (mais non obligatoire)

A. Hunt and D. Thomas.
The Pragmatic Programmer—From Journeyman to Master.
Addison-Wesley, 2000. (En vente à la COOP-UQAM.)

Références
Quelques livres particulièrement intéressants


K. Beck (2003).
Test-Driven Development—By Example.


J.A. Campbell and P.P. Papatreou (2013).
SonarQube In Action.


J. Humble and D. Farley (2011).
Continuous Delivery—Reliable Software Releases Through Build, Test, and Deployment Automation.


A. Hunt (2008).
Pragmatic Thinking & Learning—Refactor Your "Wetware".


J. Rasmusson (2010).
The Agile Samurai—How Agile Masters Deliver Great Software.


J. Richardson and W.A. Gwaltney.(2005).
Ship it ! : A Practical Guide to Successful Software Projects.


J.F. Smart (2015).
BDD In Action.


V. Subramaniam and A. Hunt (2006).
Practices of an Agile Developer—Working in the Real World.


J. Visser (2016).
Building Maintainable Software—Ten Guidelines for Future-Proof Code (Java Edition).

Références

Biblographie plus détaillée (livres)

http://www.labunix.uqam.ca/~tremblay/MGL7460/Liens/references.pdf

QUESTIONS ?


BANG!
STONE AGE
BANG!
BRONZE AGE
BANG!
IRON AGE
BANG!
DARK AGE
BANG!
MODERN AGE
BANG!
COMPUTER AGE
TAK