

# Software Evolution: A Road Map

Václav T. Rajlich  
*Department of Computer Science*  
*Wayne State University*  
*Detroit, MI 48202*  
*vtr@cs.wayne.edu*

## Abstract

*Software change is the basic building block of software evolution, and problems of software evolution can be solved by research of software change. There is a need to develop reactive methodologies, tools, and techniques. This will solve the problems posed by unexpected changes and also address the issues of ultra-fast software evolution.*

## Introduction

Software evolution is the stage in software lifecycle where substantial changes are made. The elementary building block of software evolution is software change. The research aiming at individual software change has a chance to solve problems of software evolution as a whole.

## Anticipatory vs. Reactive Approaches

*Anticipatory* approaches to software evolution recommend to predict the future evolutionary changes and to prepare the software for them in advance. One widely quoted variant recommends creating classes that will localize and contain future changes. Another variant recommends to refactor the code in anticipation of future changes.

I believe that under certain specific circumstances, we are able to anticipate and prepare for the obvious. In our daily life, we can buy car insurance before a car accident, or we can pack an umbrella before it rains. However, all anticipation is based purely on the extrapolation of past experience. Whenever faced with true surprises, we are always unprepared and forced to rely on pure luck. For a software engineer who evolves point-of-sale systems, a new state law that exempts certain approved nonprofit organizations from sale taxes on certain specific items is such a surprise. Software modification will require a substantial extension of both the database and the code.

Anecdotal experience says that software engineers face unexpected changes very often. For these situations, they need *reactive* methodologies that do not make too many assumptions and do not require a crystal

ball. Unfortunately at this moment, reactive software change is largely a self-taught art and programmers have to learn it by trial and error, a very expensive type of education.

In order to improve the situation, software engineers need research in all facets of software change. First they need a refined classification of changes (incremental changes that add functionality, refactoring changes that preserve functionality but improve the structure, changes that retract functionality). For major classes of changes, they need research in the processes and sub-processes of the change. Examples are techniques of concept location, which help to find where in the program the change is to be made. Another example is change propagation: When one component of software is changed, all interactions of that component with its neighbors may need readjustments. But after changing the neighbors, their neighbors may also need change, and so forth. Program comprehension is a necessity throughout the whole process of change.

This is a large agenda but one that is vital for the practice of software evolution.

## Speed of evolution

Demands on the speed of software change are increasing. Internet applications demand change in hours, otherwise business opportunities are lost. Gone are the happy old days when the next software delivery was several years away.

Experience shows that most of the fast changes belong to the category of unanticipated changes, because there was never time to prepare for the future anyway. Hence the research agenda of unanticipated changes supports ultra-fast evolution also. However a novel difficulty is the heterogeneity of the media and technologies that are characteristic of recent Internet applications.

## Conclusions

Software evolution needs methodologies for software change, of the same style as methodologies for new program development. In this position statement, we listed some aspects of these future methodologies.