Kari Aliranta

Implementing source space analysis functionality in a GUI application for MEG data analysis

Master's Thesis in Information Technology

October 4, 2018

University of Jyväskylä

Department of Mathematical Information Technology

Author: Kari Aliranta

Contact information: kari.p.aliranta@student.jyu.fi

Supervisor: Tapani Ristaniemi

Title: Implementing source space analysis functionality in a GUI application for MEG data analysis

Työn nimi: Lähdemallinnustoiminnallisuuden toteuttaminen graafiseen MEG-data-analyysisovellukseen

Project: Master's Thesis

Study line: Ohjelmistotekniikka

Page count: 94+0

Abstract: Meggie is a graphical frontend application for MNE, a set of Python and C tools for MEG data analysis. The application is designed to facilitate analysis as performed by students and researchers of neuroscience primarily at the University of Jyväskylä. As the application is published under a free software licence, more general usage is also a possibility.

This thesis focused on implementing source space analysis functionality in Meggie and evaluating the solutions to realize this functionality. As the the main goal of the thesis was to fulfill an existing need with a software artifact, the evaluation was performed within the paradigm of design science. Along with the benefits to the users of the application, the thesis contributes to the general discussion about the software development practices in the field of computational neuroscience, offering documented experiences about the viability of solutions. Details regarding the domain of neuroscience and the architecture of the application are provided.

Keywords: MEG, MNE, design science, graphical user interface design, software architectures

Suomenkielinen tiivistelmä: Meggie on graafiseen käyttöliittymään pohjautuva sovellus, jonka tarkoitus on helpottaa komentorivipohjaisen MNE-kirjaston käyttöä. Sen kohdekäyt-

täjiä ovat eritoten Jyväskylän yliopiston neurotieteiden tutkijat ja opiskelijat. Sovelluksen lähdekoodi on avointa, joten sen leviäminen myös laajemman käyttäjä- ja kehittäjäkunnan tietoisuuteen lienee mahdollista.

Tutkielman tavoitteena oli lähdemallinnukseen liittyvän toiminnallisuuden lisääminen Meggieen sekä tämän toiminnallisuuden dokumentointi ja toteutusratkaisujen arviointi. Tutkielman tieteelliseksi paradigmaksi valikoitui suunnittelutiede (design science), jonka mukaisen dokumentoinnin ja arvioinnin erityiskohteita olivat arkkitehtuuri, koodin laatu ja sovelluksen käytettävyys. Tutkielma kommentoi osaltaan sovelluskehitykseen liittyvää keskustelua erityisesti laskennallisiin neurotieteisiin liittyen, ja toivottavasti myös hyödyttää tuota kehitystä esittelemällä mahdollisia toteutusratkaisuja alueelle tyypillisiin kehitysongelmiin.

Avainsanat: MEG, MNE, suunnittelutiede, käyttöliittymäsuunnittelu, sovellusarkkitehtuurit

Glossary

Signal averaging	A method for preserving the common elements in signal periods to be averaged, while the irregular background noise is cancelled out.
BEM	Boundary element method. A computational method of solving linear partial differential equations.
Canopy	A scientific Python environment developed by Enthought Inc.
ECG	Electrocardiography. A method for recording electrical activity of the heart.
EEG	Electroencephalography. A method for recording electrical activity of the brain via electrodes placed on the scalp.
Epoching	Cutting the measurement data into slices based on time.
fMRI	Functional magnetic resonance imaging. A method for investigating the brain activity by measuring the blood flow in the brain.
MEG	Magnetoencephalography. A method for examining brain activity, based on detecting magnetic fields created by the electrical activity in the brain.
MNE	A software package for processing MEG and EEG (electroencephalography) data. Also short for Minimum Norm Estimate, a method for trying to solve the inverse problem in neuroscience.
MRI	Magnetic resonance imaging. A medical imaging technology based on applying strong magnetic fields upon tissues to be imaged.
MVC	Model-View-Controller. A software pattern for separating user interface logic from the rest of the application logic.
OS	Operating system.
Preprocessing	Removal of unwanted disturbances from the measurement data.
PyQt	Python bindings for the Qt application framework.

Qt	A collection of libraries, written in C++, that allow for platform-independent abstractions for various types of functionality, e.g. networking, graphical user interfaces, threading, XML.
SQUID	Superconducting Quantum Interference Device. A sensor type commonly used for measuring cerebral magnetic fields in MEG research.
Terminal	An interface to access operating system in Linux, by entering written commands and observing their output. Compare to Command prompt in Windows.
TFR	Time-frequency. In signal processing, a representation of signal over both time and frequency at the same time.
UI	User interface. Allows the user to interact with the software. In Meggie, refers to graphical user interface specifically.

List of Figures

Contents

1 Introduction

MEG (magnetoencephalography) is a method of measuring magnetic fields caused by electrical activity in the brain. By analyzing the data accumulated by the measurements, the type and location of the brain activity can be discerned. In a common type of MEG experiment setting the test subject is given sensory input, i.e. shown pictures or played sounds to, and the activity of the subject's brain is measured by the measuring equipment. Regardless of the setting, the data accumulated by the equipment needs to be preprocessed and analyzed in order to yield useful information.

The thesis introduces an existing MEG data processing and analysis application Meggie, originally developed as a student project for the Center of Interdisciplinary Brain Research (CIBR) in the University of Jyväskylä. The goal of the application was and is to make the basic analysis of MEG data easier and less error-prone, especially for the inexperienced analysts. The students and researchers in CIBR are considered the main target users of the application, and represent varying levels of expertise in MEG data analysis. The design and implementation is largely based on the requirements and feedback provided by these readily available users.

Meggie is primarily a graphical frontend for MNE-Python, a set of Python tools for MEG data analysis. In its current form, the application allows data preprocessing, i.e. removal of various types of outside disturbance from the data, and sensor space analysis, i. e. analyzing data produced by the sensors without matching it with an MRI picture of the brain. However, in order to be of wider use, the application should allow performing basic data analysis in source space. This would require matching the sensor data with actual anatomical areas of the brain. During the thesis, the Meggie will be developed further to enable a subset of this source space analysis capability.

1.1 Research problem, research methods and contribution

As explained above, the main goal of the thesis is the design and realization of a software application that fulfills an existing need. Therefore, the chosen paradigm for the research is

design science, the guidelines for which are presented by Hevner and Chatterjee (see Hevner et al. 2004, p. 82). The requirements of the software application - or "software artifact", as Hevner and Chatterjee put it - represent various sides of the problem the software artifact is intended to solve, and in the context of this thesis are considered research questions. The solutions to fulfill these requirements, then, represent the possible answers. The existing knowledge base, such as research related to software architectures and user interface design, is utilized to guide the design process and to evaluate the validity of the resulting solutions. The findings discovered during the process are contributed back to the knowledge base in the form of this thesis, thus differentiating the process from routine design.

The main scientific contribution of the thesis is, however, mainly of practical variety. As the aim of the Meggie application is to make the basic analysis of MEG data easier, the new features implemented will hopefully further facilitate the first steps of the students on the field, while also offering some time-saving tools for the more advanced. Also, the process of implementation may uncover shortcomings in the main backend used, the MNE-Python library and, via questions and bug reports, help advance its development. Furthermore, the thesis contributes to the general discussion about the software development practices in the field of computational neuroscience, offering documented experiences about the viability of solutions.

1.2 The structure of the thesis

Chapter 2 includes background information about MEG research and current software used to perform it. It also explains the structure and background of the Meggie application. Chapter 3 introduces the methodology and includes theoretical background about design science, architectural and user interface assessment and other relevant areas of software evaluation. Chapter 4 lists the requirements the development is intended to fulfill and classifies these requirements into categories. Chapter 5 documents the actual implementation and applies methods introduced in chapter 3 in order to evaluate the solutions aimed to fulfill the requirements for the software. Chapter 6 concludes the thesis and summarizes the most important findings.

2 Overview

This chapter includes background information about MEG, the role of software in analyzing data generated by MEG equipment, and Meggie as an example of an application designed to aid in the analysis.

2.1 Magnetoencephalography in brief

Magnetoencephalography (MEG) is a neuroimaging technique based on detecting weak magnetic fields caused by the electrical activity of neurons (nerve cells) in the brain (Hari and Salmelin 2012, 387; Hari, Parkkonen, and Nangini 2010, 89). This section is an introduction to MEG theory, instrumentation, data acquisition and preprocessing, and initial analysis of preprocessed data without combining it with anatomical information acquired via MRI (see below). These subjects are examined rather briefly in a later section is dedicated to matching the MEG and MRI data, also as source modeling, see section 2.2.

2.1.1 MEG basics as compared to EEG and (f)MRI

MEG is one of the various ways of detecting brain activity. It differs from another common detection method electroencephalography (EEG) in that it doesn't detect electrical activity directly, thus circumventing the distortion caused by the electrically highly resistive skull; additionally, MEG and EEG have different optimal areas and angles of detection relative to the brain and its surface, making the methods complementary to a degree (Hari and Salmelin 2012, 387; Hari, Parkkonen, and Nangini 2010, 95).

MEG also differs from another detection method, magnetic resonance imaging (MRI) and its functional counterpart fMRI, in that it doesn't use external magnetic fields as an integral part of the detection technique - in fact, external magnetic fields are an important source of disturbance in MEG measurements, warranting the installation of MEG equipment in magnetically shielded rooms (Hansen, Kringelbach, and Salmelin 2010, 45). Another differing variable between fMRI and MEG is latency: MEG can detect variations in brain activity within millisecond timeframe (Hansen, Kringelbach, and Salmelin 2010, 54), whereas fMRI, being

based on detecting brain activity via blood flow and blood oxygen levels, has a latency of several seconds at worst case (Bandettini 2009, 2; Mitra and Pesaran 1999, 692); as for the spatial fidelity, the fMRI is accurate to (at least) mm-scale, with tradeoffs in temporal resolution possibly giving more spatial accurary (Hari, Parkkonen, and Nangini 2010, 95; Mitra and Pesaran 1999, 692). In MEG, on the other hand, the spatial accuracy depends on the quality of the solutions to the inverse problem posed by mapping signal data to actual brain regions, and remains a point of debate (Mitra and Pesaran 1999, 693).

Despite radically different principles of operation, the non-functional MRI and MEG are highly complementary, as separately acquired MRI images are used when mapping MEG sensor-measured data to actual anatomical areas of the brain - a process called source modeling, explained more elaborately in section 2.2 (Hansen, Kringelbach, and Salmelin 2010, 50). MEG and functional MRI can also yield useful information when used in tandem, see for example Hari, Parkkonen, and Nangini 2010, page 95.

2.1.2 MEG equipment

A typical set of MEG measuring equipment consists of three essential parts: sensors to be placed around the head of the subject; a magnetically shielded room to protect the sensors againts disturbance from external magnetic fields; and computers and data storage equipment to store the data acquired via the sensors.

The sensors are commonly of the SQUID (Superconducting Quantum Interference Device) type. The SQUID sensors are sensitive enough to detect the minute magnetic fields created by the electrical activity in the brain, but unfortunately require heavy cooling to maintain superconductivity (Hansen, Kringelbach, and Salmelin 2010, 28). This is generally achieved with liquid helium, which in turn requires a system to insulate the cooled sensors from the scalp of the person under examination (Hansen, Kringelbach, and Salmelin 2010, 36).

The sources present two subtypes of SQUID sensors. Simpler *magnetometer* sensors are sensitive to magnetic fields both near and far, whereas *gradiometer* sensors include a compensation coil to deduct the external magnetic fields, making them less sensitive to external interference. These sensors, then, can be grouped and combined in various ways - Meggie

has been tested with data accumulated by MEG equipment with sensor triplets, each including two gradiometers and one magnetometer. (Hansen, Kringelbach, and Salmelin 2010, 32)

A magnetically shielded room is, as stated above, required to prevent the external magnetic fields from influencing the measurement. Traditionally, the these rooms have been very heavy and costly, but as demonstrated by De Tiège, more lightweight solutions may also be an option, at least in some experiment settings. (Hansen, Kringelbach, and Salmelin 2010, 45-47; Tiège et al. 2008)

The file storage space required by MEG data is in the order of gigabytes per measurement session of a single subject[1], and this is just for storing the raw measurement data; also the computing power needed for the processing and analysis of the accumulated data is high - MRI image reconstruction (see section 2.2) can take hours, while some steps in source analysis (when performed by MNE-Python methods) can require tens of minutes to complete on a single modern high end desktop computer.

2.1.3 MEG experiments and data acquisition

There are a few common types of MEG experiment settings. A setting often used as an MEG example is *trial-based*, i.e. includes a subject who is subjected to short bouts of sensory stimuli, e.g. shown pictures to, usually repeatedly. In this setting, the interesting parts of the data are the periods starting right (a few milliseconds) before the stimuli (or *events*, from the point of view of the data), and ending a short time (typically a few seconds) after it. These periods are called epochs, and are generally specifically extracted from the data, with rest of the data discarded as not relevant for the experiment. (Hansen, Kringelbach, and Salmelin 2010, 90; Gross et al. 2013, 353)

Another exemplary experiment setting is based on a continuous, longer stimulus, e.g. a clip of a piece of music. In this setting, the data is generally not split into periods around precisely given stimuli, like in the setting above, but examined with various other methods more

1. As exemplified by the official MNE sample data, with a length of 4.5 minutes, which takes 128 megabytes of space

suitable for detecting changes in oscillatory brain activity, such as those based on spectral analysis and statistical analysis of the signals. (David, Kilner, and Friston 2006a, 1580-1581)

Be the setting repeated or continuous, there are at least three kinds of neuronal responses to stimuli, requiring different analysis methods. Evoked responses are characterized by oscillations that are phase-locked to the stimulus, and can be thus discovered by averaging epoch data corresponding to single trials. Induced responses, on the other hand, have a varying latency in relation to the stimulus onset, and require spectral (frequency or time-frequency) transform for discovery, usually followed by averaging (Tallon-Baudry and Bertrand 1999, 152, 154). The third kind, the steady state responses, which are "evoked responses whose constituent discrete frequency components remain constant in amplitude and time over an infinitely prolonged time period" (Lins and Picton 1995, 420), can be subjected to frequency analysis, too (Lins and Picton 1995, 422).

The analysis pipelines most relevant to Meggie are roughly presented by the following figure, the phases of which are explained in the sections afterwards:

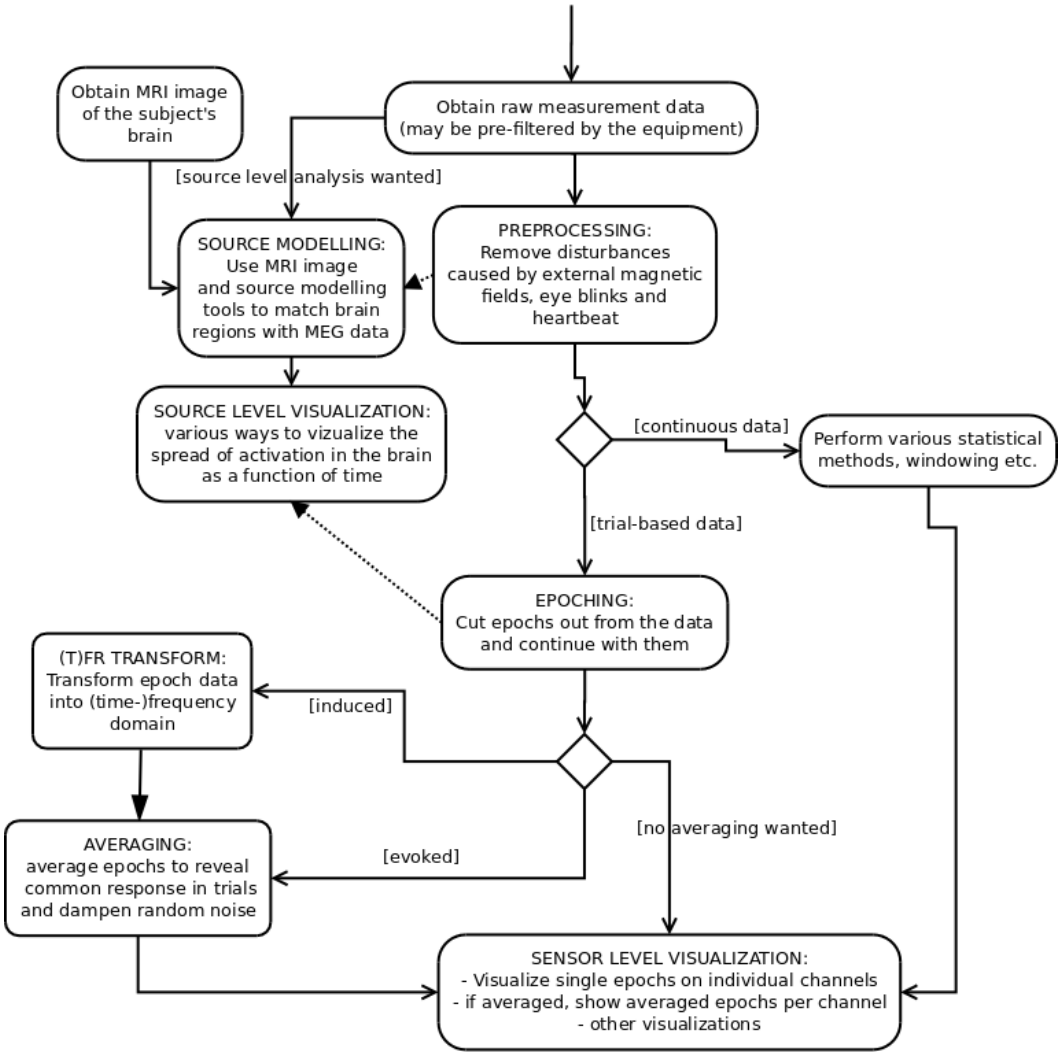

Figure 1. MEG process

2.1.4 Preprocessing

The purpose of the preprosessing phase is to remove unwanted interference from the signal data, while retaining as much of the useful data as possible. According to Gross et al, there are three categories of interference (or artifact) sources: system related sources, i.e. the interference originating from the system itself, e.g. broken sensors; external sources, such as nearby power lines[2], electric appliances and mechanical vibrations; and physiological sources, which include head movements, heartbeats and eye blinks. (Gross et al. 2013, 352)

Gross et al. also give recommendations on handling these interference types. Recommended methods for removing system-related interference include excluding obviously bad channels and data epochs by simple visual inspection or automatic rejection based on preselected parameters, (notch) filtering at the frequency of the interference and ICA, Independent Component Analysis. For physiological artifacts, Gross et al. recommend same methods as for system artifacts, with reference sensors and following interference subtraction for eye blinks, heartbeat and other possible sources. (Gross et al. 2013, 352)

For removal of external artifacts, we also have to turn to Hansen, Kringelbach and Salmelin. They, too, mention reference sensors, in this case placed outside the MEG sensor helmet for measuring the external magnetic fields. More recent and sophisticated methods include signal-space separation (SSS) and signal-space projection (SSP), both of which employ a practical presentation of the mathematical concept of *signal space*, separating the overall signal data into subspaces of data originating from the brain and interfering external data. For details of the methods, see Taulu, Kajola, and Simola 2003 and Uusitalo and Ilmoniemi 1997 - for purposes of Meggie and source modeling, it is enough to know that the data has been preprocessed with one of these, or equivalent, methods. (Hansen, Kringelbach, and Salmelin 2010, 48)

2. Gross et al. have actually classified line noise to both system related and external sources; I choose to put it here on the basis of electric appliances being here, too

2.1.5 Epoching

Epoching is technically a rather straightforward procedure involving selection of the desired portions ("epochs") of the data for further analysis. The exact parameters (such as the number of trials, start and end times, length, and time between each epoch), however, depend on the research questions at hand and require careful planning. Gross et al. list various caveats related to the parameters. These caveats mostly come into play when averaging the epochs, which is handled in the next subsection.

It should noted that epoching as a concept is usually related to the analysis of event related fields (or ERFs) and is mostly used when discussing trial-based experiment settings. There are, however, other circumstances where the data may be handled in small portions, despite the setting not being trial-based – for example when performing windowing in algorithms for spectral analysis (see Gross et al. 2013, 355).

2.1.6 Averaging

Signal averaging of time-locked trials is a common and rather simple method of discerning actual signal data from noise. It is performed by computing average value of data points in trials, assumed to include consistent signal components and having same timing. It also assumes that the noise in the measurements is (sufficiently) random and that the signal and noise components are (sufficiently) uncorrelated. Given these preconditions and enough trials, actual signal data with a significantly lower magnitude than the included noise can be discerned. (Drongelen 2007, 55-56)

As for the non-time-locked trial data, the simple averaging is not appropriate, as the temporally varying responses may cancel each other out. In these cases, the typical solution is to first convert the signal data epochs individually into time-frequency domain and then average the epochs to reveal the average power of the signal. (David, Kilner, and Friston 2006b, 1580)

The simple time-locked averaging is a staple of examining evoked responses, where averaging of epochs is common way of increasing signal-to-noise ratio, despite caveats provided by the recent research (Gross et al. 2013, 353-354). Time-frequency analysis, on the other

hand, is commonly used when analyzing induced responses. Again, there are caveats against simplistic classification of responses to purely induced or evoked (see David, Kilner, and Friston 2006b), but for the purposes of features of Meggie, the classification to time domain and time-frequency domain analysis is sufficient.

2.1.7 (Time-)frequency transformation and analysis

In addition to simple averaging, the transformation of the signal data into frequency or time-frequency domain is a common practice in neuroscientific data analysis. It is motivated by the fact that oscillatory signal components are more easily detectable in those domains. The methods of transformation can be broadly classified into parametric and non-parametric, the latter of which include fast fourier transform and various wavelet transform methods, among others. The methods provided by Meggie include only non-parametric ones. (Gross et al. 2013, 355)

2.1.8 Sensor level visualization

The sensor level visualization refers to visually representing signal data that has not yet been converted to source domain, i.e. has not yet been linked to actual brain regions with source modeling methods. Despite this lack of conversion and the solution for inverse problem, possibly meaningful information can be gleaned from the data due to the fact that the sensors most readily measure the field strength in the nearest brain regions, and the location of the sensors is known (Gramfort et al. 2013, see chapter about linear inverse methods). Visualizations provided by Meggie include graphs for single sensor activity, sensor topography aligned by brain regions and channel averages from selected channels, all expressed as signal strength / time.

2.2 Source modeling in MEG research

The goal of source modeling (or source analysis, or source localization, or source reconstruction) is to determine the actual neural sources of the magnetic signals measured by the sensors. The source modeling consists of two essential steps: the construction of the forward

model, which describes the magnetic field distribution created by a single source of known location and orientation; and finding the solution for the inverse problem, which requires deciding which of the several (potentially practically infinite, but reduced by constraints) possible configurations of field sources is the closest match to actual activity in the brain. (Gross et al. 2013, 356).

There are various methods of source localization, broadly classified into three categories: parametric, scanning and distributed inverse. The background library MNE and therefore, potentially, Meggie, provides methods that fall under the latter two categories (Gramfort 2013, 5). Also, in all methods provided by the MNE-Python library and the older MNE-C scripts, the elementary source is assumed to be a current dipole, and large number of these dipoles are assumed to cause the overall magnetic field measured by the MEG equipment. The minimum-norm estimates (MNE) are then used to decide between the dipole configurations that would explain the measurements equally well – along with the constraints provided by analyzing MRI images, in combination with general knowledge about directions of the currents caused by active brain cells (Gramfort et al. 2014, 3-4)

Regardless of the actual method used, there are common steps in source modeling that must be taken before a solution to the inverse problem can be said to have been found. The source modeling process can be roughly presented by the following figure

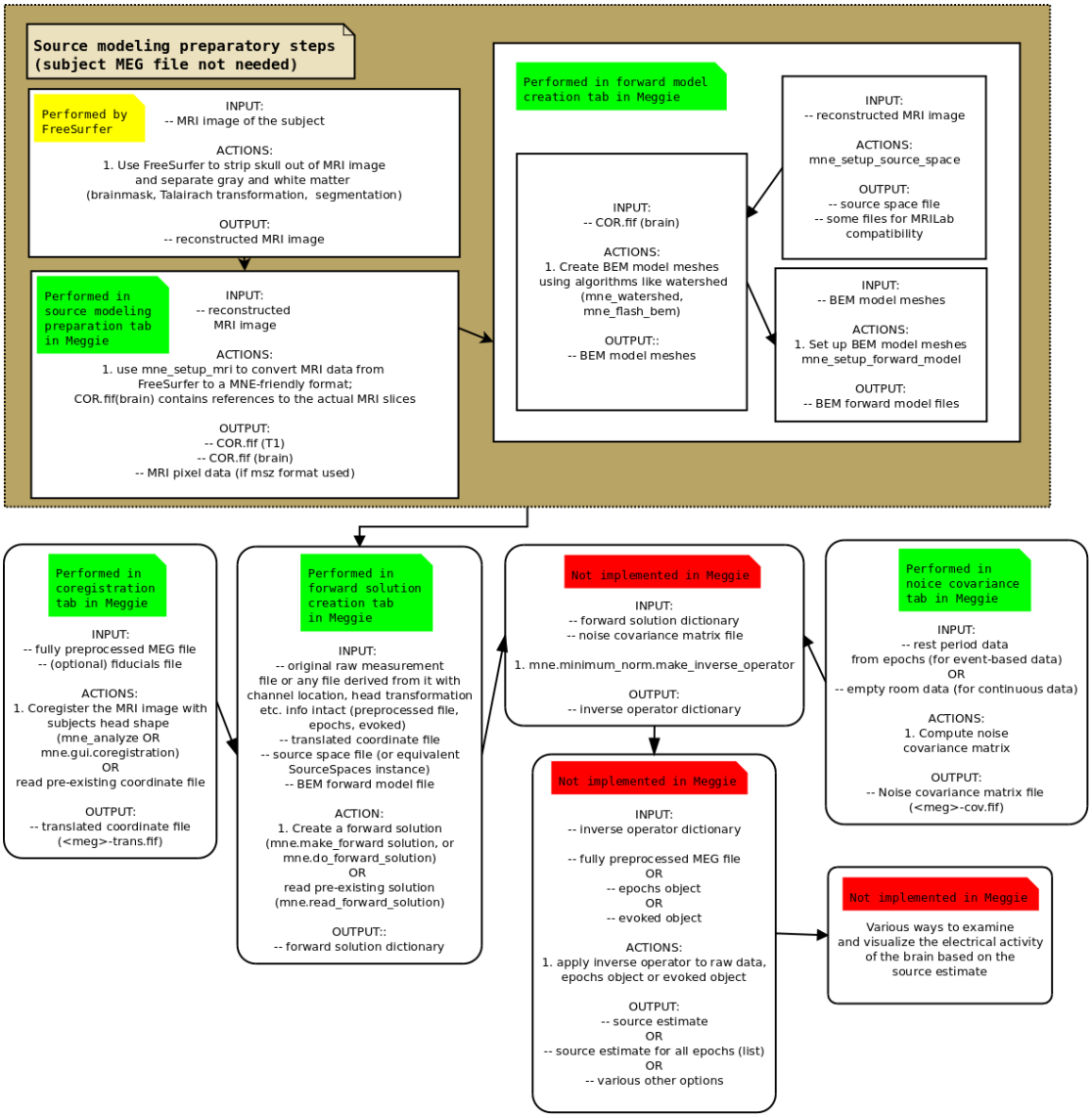

Figure 2. Source modeling process

Regarding the process above, the following should be noted:

- The ordering of the steps is not always set in stone. The creation of the noise covariance matrix, notably, can be performed irrelevant of the rest of the steps. The same holds for coregistration, which only requires a fully preprocessed MEG data file and the separately acquired MRI data for the subject. See respective subsections for details.
- The green and red notes about the functionality implemented in Meggie describe the state of the application after the thesis work. They are relevant for the section 5.2 which details the evaluation and implementation of the source modeling features.

2.2.1 MRI image reconstruction

Also known as cortical surface reconstruction. This phase takes MRI image data as its input, and outputs data with non-cebebral tissue (e.g. skull) removed, white and gray and pial matter separated and anatomical brain regions labeled. This data is later linked to MEG measurement data in a process called coregistration, see below. (Dale, Fischl, and Sereno 1999, 180; FreeSurfer Team 2017)

Based on expert user comments, the phase can take hours to complete even on a modern PC running the FreeSurfer MRI image processing suite[3]. The phase has therefore been omitted from Meggie, which starts its source modeling pipeline by importing the FreeSurfer pre-reconstructed data files.

2.2.2 MRI image data description

According to MNE manual, the mne_make_cor_set executable, which is called by mne_-setup_mri script, "creates a fif format MRI description file optionally including the MRI data using FreeSurfer MRI volume data as input". Based on the description of the method, no actual new data is generated. This phase is needed for MNE to understand the MRI data recontructed by FreeSurfer.(MNE Developers 2016)

3. The FreeSurfer developers also do warn about the computing power requirements of the reconstruction workflow, see FreeSurfer Team 2008

2.2.3 Forward model creation

The forward model, as stated is earlier, describes the magnetic field distribution created by a single source of known location and orientation. In Meggie, creating the forward model actually includes three separate steps, but as Meggie groups the steps into a single one UI-wise, they are handled under a common header.

Source space setup

Setting up the source space involves specifying the locations of candidate dipole sources. There are various methods of doing this, but the specification method normally used by MNE samples the candidates on the boundary between the white and gray matter. According to Gramfort et al. in 2013, to reduce the number of possible dipoles without losing too much information on the surface topology, MNE uses subdivided polygons when setting up the source space. With default settings and typical subjects, this results in average spacing of around 3 mm between the candidate dipoles. (Gramfort 2013, 5)

Apparently, however, the MNE defaults for the source space setup have changed since the publication of the above article. The MNE manual (see MNE Developers 2015) refers to the default method as "traditional" and suggests a spacing of 7 or 5 mm with this method. Polygonal methods, on the other hand, are declared optional. Regardless, the spacing of the dipole candidates, being projected on a 3-D space, greatly affects the computation time of the later step in the process, the forward solution creation.

BEM model meshes creation and model setup

Creation of the forward model also requires head model, which is "literally a model of the influence of the head geometry, and magnetostatic properties of the head tissues, on magnetic fields measured outside the head" (Hansen, Kringelbach, and Salmelin 2010, 109). The model is based on an MRI image, and there are several computational methods for creating it (Hansen, Kringelbach, and Salmelin 2010, 109-110), of which MNE uses the BEM (Boundary Element) method. The BEM meshes creation consists of geometrical tessellation of the MRI image - for this, MNE provides two methods, of which the default one is based on triangular tessellation with watershed algorithm. (Hansen, Kringelbach, and Salmelin 2010,

110-111; MNE Developers 2015). Also, the model creation requires estimating the (electrical) conductivity values of the brain, conventionally based on separate studies. (Hansen, Kringelbach, and Salmelin 2010, 112-113)

2.2.4 Coregistration

The forward solution creation requires defining the boundary-element surfaces, the source space and the MEG sensor locations in a common coordinate system. In MNE, this is performed by relating the MRI coordinate system used by FreeSurfer and the MEG coordinate system. The latter system is defined by the fiducial landmarks, i.e. two pre-auricular points below the ear canal and a single point in the nasion. The sensors locations are then aligned with these points with the help of the carefully positioned Head Position Indicator (HIP) coils, which generate magnetic fields for the sensors to measure and thus locate and follow during the measurement - otherwise any movements of the experiment subject's head within the measurement equipment would cause errors in locating the sources of the measured magnetic fields. To finalize, an algorithm to refine the initial manual estimate of the fiducial points is run. (Gramfort 2013, 8; Hansen, Kringelbach, and Salmelin 2010, 50)

2.2.5 Forward solution creation

In MEG measurements, the creation of the forward solution requires three things: an approximation of the electrical and magnetic properties of the head (computed in BEM model setup step), the specifications of the source space (computed in source space setup step), and the configurations of the MEG sensors (specified in coregistration step). (Gramfort 2013, 7) Once all of these are available, the forward solution can be created. The solution creation in MNE requires few user inputable parameters, the most nonobvious of which is the minimum distance of assumed sources from the inner skull surface (MNE Developers 2015).

2.2.6 Computation of the noise covariance matrix

The noise covariance matrix is needed for noise removal and could also be classified as part of the preprocessing step (see page 8). For the purposes of the thesis work, it is considered

a source modeling step since the first MNE method that requires it is the inverse operator creation (see below) (MNE Developers 2015).

The computation can be based on a time interval of raw data or on already epoched data. In the case of raw data, the recommended choices are an empty room without a subject to get the noise profile of background magnetic fields only, or the with the subject before the start of the stimuli; in the case of epochs, the data from time intervals before starting the stimuli are recommended. (MNE Developers 2015) As for the method, the MNE implementation uses Independent Component Analysis (ICA), and it can be useful when used in conjunction with SSS and SSP methods detailed in the section about preprocessing (Gramfort 2013, 6-7).

2.2.7 Inverse operator creation and source estimate

The inverse operator is an operator that maps the (sensor) signals to source space, or more technically, can be used to multiply the original measured (sensor) data matrix, resulting in a source estimate. (Hauk, Wakeman, and Henson 2011, 1967-1968; Schoffelen and Gross 2009, 1862). In MNE, the algorithm is a little more complicated than that (see MNE Developers 2018b), and the method for inverse operator creation takes the forward solution and the noise covariance matrix as parameters, resulting in an inverse operator. The operator can then be applied to either the raw file, an epoch collection or an evoked dataset. In all cases, the final result is called source estimate, which can be analyzed in various ways (MNE Developers 2015, MNE Developers 2018b).

2.2.8 Source estimate analysis

The source estimate is "the estimated spatial distribution of the neural (primary) currents given the MEG measurements" (Hansen, Kringelbach, and Salmelin 2010, 50), i.e. it contains an estimation about the locations of electrical brain activity that was measured with MEG equipment - indirectly via magnetic fields caused by that activity, as described in section "Magnetoencephalography in brief" on page 3.

The analysis of the source estimate is also concerned about the temporal dimension of the brain activity, i.e. how the locations of neural activation change as a function of time

(Hansen, Kringelbach, and Salmelin 2010, 87). The change of activation could simply be presented as still images or movies of the brain from various angles in 3D or with 2D slices; or there may be further quantitative analysis of the activation, with associated graphs and plots (MNE Developers 2015).

An example about further examination of the source estimate is connectivity analysis. The connectivity analysis is concerned about the cooperation of brain structures, as represented by the temporal syncronization of the activation in different brain regions in reaction to the task defined by the set of stimuli in the experiment. Determining the connectivity requires employing various statistical methods, the results of which could then be visualized with various kinds of plots. (MNE Developers 2015; Schoffelen and Gross 2009, 1857-1858; Hansen, Kringelbach, and Salmelin 2010, 216-217)

2.3 About Meggie

Meggie was started as a student project in the Department of Mathematical Information Technology as the University of Jyväskylä, Finland. The aim was to facilitate the processing, analysis and visualization of MEG data, with a specific focus on novice-level usability for beginning neuroscience students. (K. Aliranta 2013, 5)

The actual student project ended with a mostly working piece of software for preprocessing and ECG / eye blink artefact removal. Additionally, basic epoching and averaging worked for the most part, and a sensor topography view for averaged channels was integrated into Meggie. At that point Meggie was, however, only capable of processing and analyzing a single subject (source file) at a time. Capability for processing multiple subjects simultaneously and performing the preprocessing and artefact removal steps in batch was implemented concurrently with the work on this thesis, by another member of the original Meggie team. There were also short periods of other student work on Meggie, concentrating mostly on making the epoching functionality more robust and enhancing the statistical analysis capabilities.

2.3.1 Functionality before the thesis work

The functionality before the thesis work has, in fact, already been detailed in the section 2.1, with caveats detailed in that section. The more significant aspect of thesis work was actually the concurrently ongoing work on the multiple file analysis capability. This required, in addition to standard software development collaboration practices, assuming that the Meggie would very soon have this capability, and taking into account its effect on the software architecture of the application. Therefore, the definition of "before" is "after the the multiple file analysis capability had already been implemented".

2.3.2 The user interface of Meggie

The user interface of Meggie consists of a single main window with various sections detailed below, and multiple dialogs that are mainly used for parameter input of data manipulation commands in each step. The main window represents the state of a single experiment at a time, with one or more subject files per experiment.

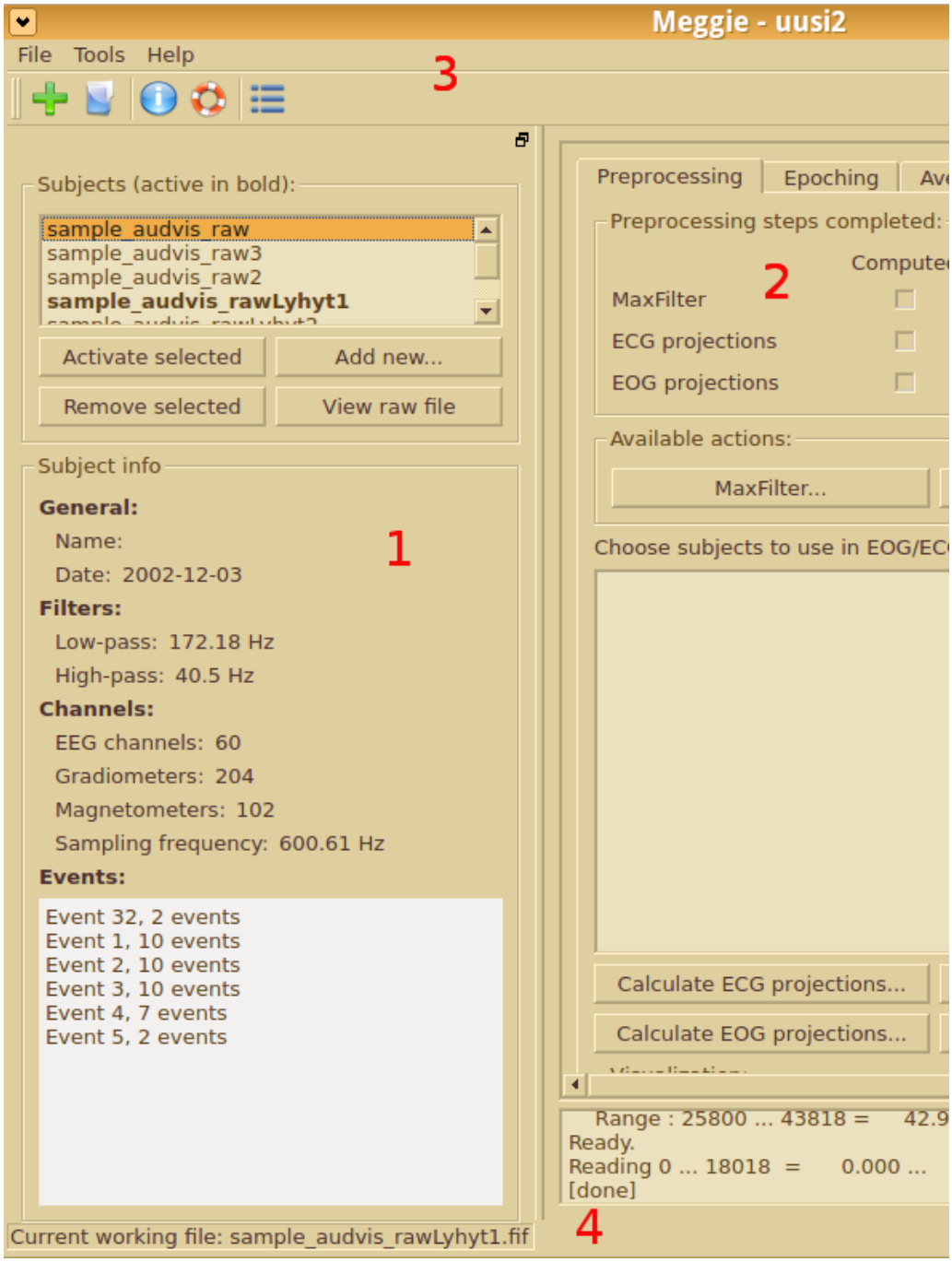

Figure 3. Meggie user interface

The sections are:

1. The left section, including the subject (raw file) list for the current experiment and the UI elements for adding, removing and switching between subjects, and showing basic info about the subjects.
2. The main analysis section, which has one tab for each process step, arranged left-to-right in the order the steps are most typically performed. The bottom part includes a scrollbox for command line output generated by Meggie and MNE.
3. The menubar and the toolbar. These mostly duplicate each other's functionality due to low amount of functionality currently available.
4. The statusbar. Only used for showing the current working file, i.e. the file which will be the target of the next process step.

3 Methodology

The principal goal of the thesis is to advance the development of a piece of software with real world use. Therefore the methods used should, preferably, aid in designing and implementing a piece of software with high quality. Additionally, to contribute to the general knowledge and to help in future creation of similar software, the methodology should make explicit the basis and value of choices made during the process of design and implementation.

The broad methodological paradigm of choice is, therefore, the design science as presented by Hevner, Chatterjee, March and Ram. Central to the design science is the concept of an artifact, which in the context of information technology may have various definitions, such as "constructs (vocabulary and symbols), models (abstractions and representations), methods (algorithms and practices), and instantiations (implemented and prototype systems)". The design science research evaluates this artifact as a solution to an organizational problem, in order to better understand the problem itself and the feasibility of the solution the artifact offers. (Hevner et al. 2004, p. 77)

In the case of the thesis, the most relevant type of the IT artifact is the instantion, i.e. the application created during the process. Other types of artifacts will, however, be used to describe the domain of the application (constructs), the application itself from various viewpoints (models) and the simple algorithms created (methods).

The organizational problem related to the artifact is harder to define. If we don't want to omit the word "organizational", we could claim that as the artifact, Meggie, reduces the manual work required for the analysis, it frees the resources (such as time) of the organization using it to be used elsewhere. As such, the problem in question would be the inefficient usage of resources in the organization.

As for the design science as research and experimentation, Hevner et al. stress the experimental and exploratory aspect of the artifacts and observe that "artifacts constructed in design science research are rarely full-grown information systems that are used in practice" (Hevner et al. 2004, p. 83). Such is the case here, too: despite not starting the creation of Meggie

from scratch, the work is still somewhat experimental and prototypal in nature, and the resulting IT artifacts will be considered valuable also in that they clarify the problems they were designed to solve. Some practical usage of the instantiation is, however, anticipated depending on the stability of the implementation.

3.1 Evaluation methods

As stated above, an integral part of design science research is the evaluation of the designed artifact. To be exact, the evaluation judges how thoroughly the design and the resulting artifact fulfills the requirements set for the arfifact (Hevner et al. 2004, p. 85). The rest of the methodology section is, therefore, dedicated to this evaluation and methods of performing it.

To keep the focus, research-based evaluation methods only used for the most important and challenging requirements. These, in the case of Meggie application, can be arqued to be equal to the requirements related to user interface and software architecture. The former are of prime importance due to the focus of the application on usability: the application does not include new analysis functionality per se, but aims make existing functionality provided by e.g. MNE-Python libraries more usable. A sanely designed software architecture, on the other hand, makes the application more modifiable, and allows for easier debugging and further development especially for other people than the original authors. Besides, there seems to be a host of research literature on both topics, divided on various schools suggesting differing solutions to similar problems.

In practice, though, the emphasis in evaluation is on guidelines and lists of best practices, even when they are are based on well-known research. Applying thorough and rigorous methods of user interface or architecture evaluation would be a subject of a work in itself, whereas this thesis focuses on finding the most important requirements for the developed software via prototyping, and hopefully fulfilling these requirements with (preliminary) solutions.

It should be noted, however, that evaluation is not something that only comes in last. Perhaps mirroring the modern agile software development trends, Hevner and Chatterjee state that "design is inherently an iterative and incremental activity": there is arguably a three-way

relationship between the design, construction and evaluation of the artifact, in that "the evaluation phase provides essential feedback to the construction phase as to the quality of the design process and the design product under development" (Hevner et al. 2004, p. 85).

According to previous experience of the thesis' author, this is indeed the case. The practical implication of this is that also the process, not only the outcome, should be documented to some degree. This helps in explaining the reasons for the conclusions and solutions, and aids in fulfilling the purpose of the work. The process documentation takes the form of a thesis diary that can be publicly seen in code commit logs. These sources are not included in the thesis per se, but it should be noted that they are used during writing of the analysis section and are likely to cause some notes about the process appear there.

3.2 User interface evaluation

The evaluation of the user interface of Meggie aims to avoid the easily avoidable mistakes by use of common heuristics, and gleans design ideas by creatively following recommended practices. The definition of usability used in the thesis is that of Jacob Nielsen: it includes such aspects as learnability, efficiency, memorability, low error rate and satisfaction of use. (Jakob Nielsen 1993, 26)

For designing and evaluating the user interface, three differing classes of sources are used if appropriate:

1. General user interface guidelines for (desktop) computing. These include various guides of standards and best practices in software user interface design. These include widely used recommendations of Jacob Nielsen and *Microsoft UX guidelines* for classical Windows (non-touchscreen) desktops (see Jakob Nielsen 1993 and MSDN 2010). Even more general, out-of-the-software-design-field usability classics such as Donald Normans *The Design of Everyday Things*, are however, not used, at least not by direct reference. Also, there are various schools for designing user interfaces from the ground up – none of these, however, are exclusively used as a basis for the design or evaluation of the Meggie user interface.
2. Specific sources related to user interface design challenges for scientific computing

software or even bioimaging software. These mostly include the sources analyzing, more generally and not limiting themselves to user interfaces, the best practices for developing said software artifacts. "Good practice for conducting and reporting MEG research" by Gross et al. is a good example of these.

3. Actual preliminary user testing. This is highly informal, is based on opinions of a few voluntary test users and concentrates on finding the points of bad usability for an inexperienced MEG analysts and making the application respond sanely to surprising user input.

3.3 Architectural evaluation

The broad definition of software architecture used in the thesis is that of Bass:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them. (Bass, Clements, and Kazman 2003, 23)

It should be noted that the separate analysis of the "externally visible properties" is performed as user interface evaluation, see above - however, the relationship of these properties to other components is analyzed as architecture analysis. Also, the architectural evaluation of Meggie intermingles with the evaluation of quality attributes, as many of them are related to code and consequently architectural quality.

The sources used for architectural design and evaluation can be classified quite similarly to those related to user interface. There are two classes of sources:

1. Sources relating architecture evaluation, design patterns and best practices for object oriented programs in general. These handle various qualities of software artifacts and architectures, and delve in more detail into such topics as class structure, separation of concerns and error handling. Firstly, though, architecture needs to described in order to be evaluated, and for this, the diagrams (especially sequence diagrams) of the classic 4+1 view model by Philippe Kruchten et al. (see Kruchten 1995) are used.

2. Python specific standards, design patterns, recommendations and PEP (Python Enhancement Proposal) recommendations. These can be readily found at Python Software Foundation website and practical programming books describing best practices with specific technologies such as PyQt.

3.4 Evaluation of other requirements

Other requirements to evaluate include requirements related to such quality attributes as performance, interoperability and robustness. These are clearly secondary requirements in an exploratory work like this, and only warrant a cursory glance.

4 Requirements

This chapter specifies the requirements the thesis work should enable Meggie to fulfill. The requirement categories follow the system presented by Karl Wieger (see Wieger 2013, 190-199), with categories not too relevant to Meggie omitted or merged with other categories.

The priorities of the requirements are classified into two categories: essential (noted by the text in the beginning of the requirement item) and non-essential (not specifically noted). Essential requirements need to be fulfilled to have a completable, if rudimentary, source analysis chain, whereas completing non-essential requirements allows for useful extra functionality. The priorities have been decided in collaboration with the client and justification for them is presented when needed.

Prior Meggie requirements not related to source space analysis are elaborated on only briefly, unless implementing new functionality requires paying specific attention to them or redefining them, or there has otherwise been considerable work related to them. The current state of Meggie is, if needed, explained in parentheses after each requirement item. If a requirement doesn't seem to be directly related to source space analysis but is a dependency for other requirement(s) that are, it is also explained in parentheses after the item.

4.1 Overall description

This section lists the requirements related to other applications used in MEG analysis (product perspective) and identifies and describes the targeted user classes of Meggie as requirements. It also specifies the intended operating environment (operating system, programming language etc.) and the resulting contraints imposed upon development by said libraries and environment. These requirements should be kept in mind while developing, but do not need separate evaluation unless there are specific problems related to them.

4.1.1 Product perspective

1. (Essential) Meggie should utilize the existing graphical user interface functionality of the MNE (Python) libraries, especially when related to raw file examination and coregistration.
2. Other MEG analysis applications should be analyzed for good solutions and where applicable, these solutions should be duplicated.

4.1.2 User classes and characteristics

1. (Essential) Meggie's main user target should be inexperienced MEG analysts, i.e. those who are not thoroughly familiar with the MEG analysis chain and possibly haven't used existing (command line) tools at all. For these users, the application should, by user interface cues and possibly other means, offer guidance towards typical analysis paths.
2. Secondary target for Meggie should be the advanced users, familiar with the current tools, who come to Meggie expecting a reduction in manual labor of the analysis process. "Manual labor" this context means such things as executing terminal commands with lengthy parameters lists and copying files around via terminal commands or graphical file managers.

4.1.3 Operating environment

1. (Essential) It should be possible to run Meggie as a non-networked desktop application on an x86 (64-bit) hardware, with Fedora 19 (64-bit) or Ubuntu 14.04 (64-bit) as the operating system.
2. Meggie should include, in an installer or otherwise, all the external libraries required for itself. Ideally, the only user installable dependency should be FreeSurfer (too large to package with Meggie), leaving e.g. a compatible version of MNE, PyQt and MNE-C scripts to be included in the Meggie package. Also, Meggie has been developed with a scientific Python distribution (Anaconda) providing plenty of packages required by MNE Python and Meggie - ideally, all of these should also be provided, as the it has

been found out before that updating these packages may very well break functionality.
3. The user-installable external dependencies (FreeSurfer and MNE-C libraries) should be easily integrated into Meggie, preferably with user not needing to run any scripts to setup their environments.

4.1.4 Design and implementation constraints

1. The new source analysis functionality implemented in Meggie should be, as much as possible, be based on existing Meggie infrastructure code and functionality. This functionality includes such things as experiment, subject and preferences handling.

4.2 System features

This section lists, as requirements, the new features to be implemented in Meggie to enable the source space analysis functionality. The features are classified according to the analysis process steps outlined in figure 2. Features without a link to a specific step are classified as general. Also, please note that any requirements related to the user interface are avoided here on purpose - they are detailed in their own sections.

4.2.1 General

1. There should be a proper system for editing and saving various preferences used by Meggie. These preferences spefically include paths to the external dependencies needed for source space analysis, i.e. MNE-C and FreeSurfer. (At the beginning of the thesis work, the only preference saved was the working directory of Meggie)
2. Meggie should automatically set the environment variables needed by the external dependencies (see above), based on the paths saved in the preferences handling system above.
3. It should be possible to automatically open a previously used experiment when starting Meggie. This in not directly related to source modeling, but is convenient for the users and especially developers constantly opening and closing the application.

4.2.2 Source modeling preparation

1. (Essential) It should be possible to import FreeSurfer-reconstructed cortical surface files into Meggie.
2. (Essential) It should be possible to trigger the mne_setup_mri script to convert the surface files into MNE-friendly format.
3. It should be possible to examine to imported cortical surfaces files for Talairach transform, skull trip, surfaces and segmentation.
4. Re-importing the cortical surface files should invalidate the later source modeling steps for the subject file in question.
5. There should be a warning and a confirmation dialog for the re-import.

4.2.3 Forward model creation

1. (Essential) Meggie should combine setting up source space, creating bem model meshes (with watershed algorithm) and setting up forward model into a single analysis step.
2. It should be possible to skip first two parts of the forward model creation by using previously created results for them.
3. Meggie should be able to create and store several forward models created with different parameters.
4. The forward model creation parameters should be saved and readable for each model.
5. It should be possible to remove previously created forward models from the Meggie data structure.

4.2.4 Coregistration

1. (Essential) It should be possible to use mne.gui.coregistration from MNE-Python for coregistration.
2. There should be an option to import a pre-existing coordinate file into Meggie.
3. Meggie should be able to create separate coordinate files for each forward model.
4. It should be possible to re-perform coregistration on an already coregistered forward model, overwriting the previous coordinate data.

4.2.5 Forward solution

1. (Essential) Meggie should be able to create a forward solution for all forward models in the Meggie data structure.
2. Re-creation of a forward solution should be possible, and in these cases the previous solution does not need to be retained.
3. The forward solution creation parameters should be saved and readable for each solution.

4.2.6 Noise covariance

1. (Essential) It should be possible to create the noise covariance matrix based on a the raw file in with Meggie experiment, or an external raw file.
2. It should also be possible to create a raw file based on an epochs collection.

4.2.7 Inverse operator

1. (Essential) It should be possible to create inverse operator based on previously created forward solutions.

4.2.8 Source estimate

1. (Essential) It should be possible to create source estimate based on previously created inverse operator and a given epoch collection.
2. It should be possible to create source estimate based on previously created inverse operator and a given evoked dataset.
3. It should be possible to create source estimate based on previously created inverse operator and a given raw file.

4.2.9 Source visualization

1. (Essential) It should be possible to perform basic visualization on brain activation as a function of time.

4.3 Data requirements

The source analysis chain processes large amount of data in various formats. This section should, therefore, list the requirements related to these formats. It should describe both the data that is fed into Meggie (input) and the data that is produced by it by the required processes (output).

In practice, though, the file formats accepted by Meggie are defined by the MNE library methods Meggie supports, and the chosen input and output parameters of those methods. Some of these methods allow for optional input or output file formats that Meggie is not required to support. Also, optionally, some analysis steps should be skippable by importing a pre-made file. All of the related requirements are, however, already described in the System Features section above. Additionally, since the purpose of the thesis is not to develop new low level data processing methods and file formats, no data requirements are listed here. For a list of file format names at each step of the analysis chain, see figure 7; for a broad description on input and output data, see section 2.1 or figure 2 on page 12.

4.4 External interface requirements

The requirements related to interfaces are elaborated in this section. In Wieger's classification, "interface" is a broad term, apparently desribing any component that relays information between different parts of the information system, be those parts user and the application (user interfaces), a software component and another (software interfaces) or the software components and hardware (hardware interfaces). Wieger also lists communications interfaces, which include such things as network protocols, email and electronic forms. Meggie doesn't make use of the latter two of these is any special way, so there are no requirements related to them. (Wieger 2013, 190-199)

4.4.1 User interfaces

1. Adding the user interface components for source analysis capabilities should not make the user interface too confusing.
2. There should be a menubar with buttons for most commonly used general functions of

Meggie.

3. Meggie preferences should be found in single location in the user interface.
4. Ongoing computations should not make the user interface of Meggie unresponsive.

4.4.2 Software interfaces

1. The interfaces between MNE, FreeSurfer and Meggie components should be clear and consistent.

4.5 Requirements related to quality attributes

These requirements relate to such things as usability, performance, modifiability, scalability, security and safety. Wieger calls for specificity and quantification of the requirements - in this case, specifity is the goal with usability and modifiability requirements, whereas quantification would be possible with performance requirements, in the form of time and memory usage limits for computation steps. There are, however, no performance requirements in the thesis work. (Bass, Clements, and Kazman 2003, 79-85)

4.5.1 Modifiability and reusability requirements

1. (Essential) Meggie should conform to MVC model or some other model of decoupling the functional code (backend) and the user interface code (frontend), making separate development, easier modifiability and greater reusability of UI components possible.
2. Meggie error and exception handling should follow the Python EAFP (Easier to Ask Forgiveness than Permission) coding style recommendation - or some other style that allows for decent error reporting to the user while keeping the error reporting code out of other program code (see below).
3. Error reporting code (e.g. creating the actual dialogs) in the backend should be avoided and relegated to the frontend. This is related to the MVC model and EAFP requirements above.
4. File reading and writing should be handled by a specific module or there should be some other way of keeping long sections of I/O code out of other program code.

5 Implementation and evaluation

This chapter describes and analyzes the implementation and the design decisions behind it. It roughly follows the order of the requirements listed in chapter 4, listing the related requirements first and then proceeding to the analysis. The analysis consists of presenting and justifying the user interface implementation first, then moving on to describe the back-end implementation with the help of diagrams when needed. The fulfillment of the overall requirements listed in section 4.1 are only evaluated as part of the other, more concrete, requirements.

5.1 Overall architecture and package structure

The implemention of preliminary source modeling capability did not, in fact, greatly alter the overall structure of the software. The package diagram depicting high level organization of the modules is the following:

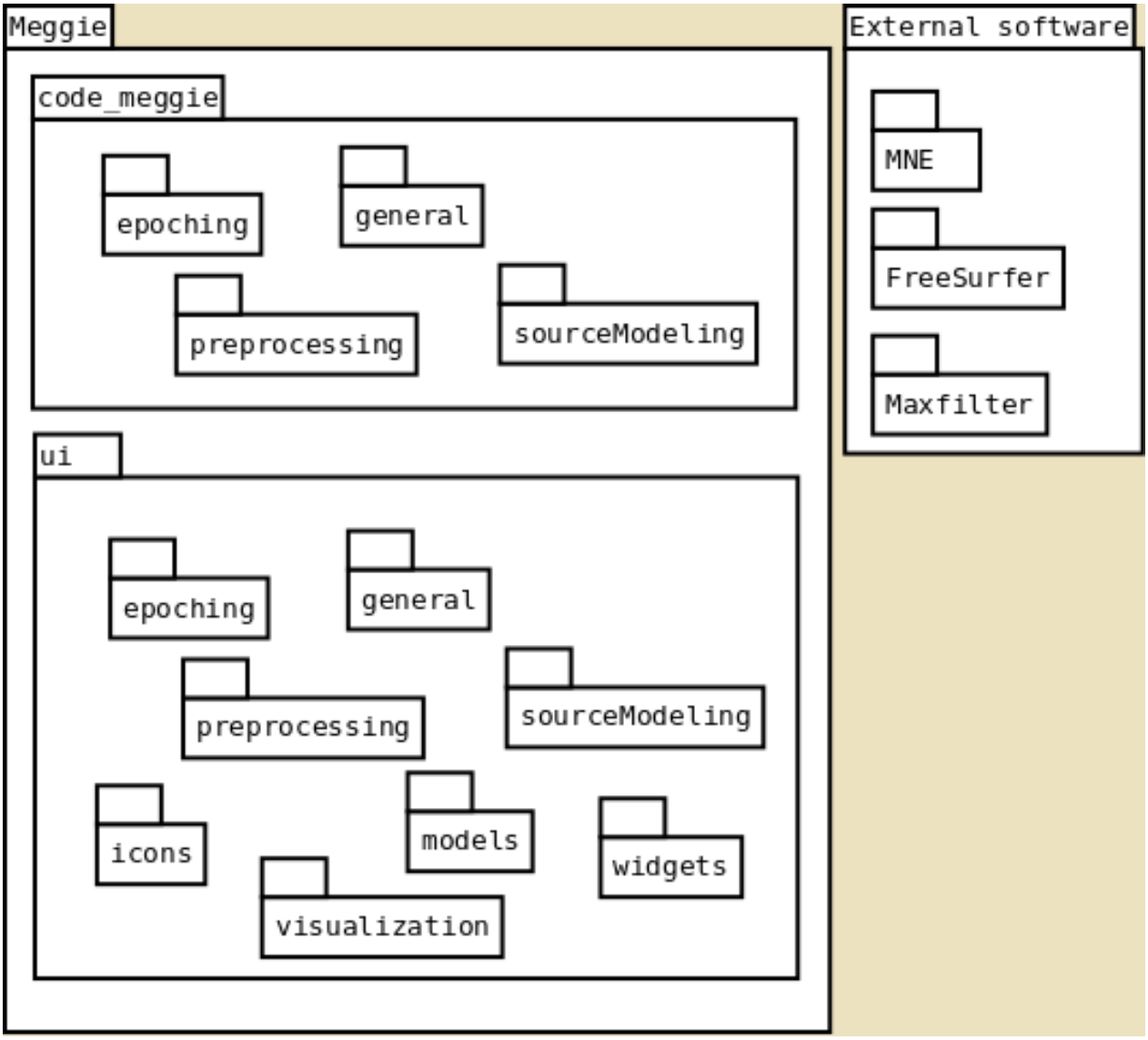

Figure 4. Package diagram of Meggie

By far the most important modules are included in the "general" packages of both "code_-meggie" and "ui" superpackages.

The general package of the "ui" superpackage includes the ui code for the main window, which, in turn, includes code for opening practically all the user interface dialogs. An earlier attempt to classify the code into several packages with clearly defined responsibilities can be seen in the "visualization" package, which includes code for input dialogs that take parameters in order to graphically present aspects of the processed data. During the thesis work, a module for realizing the MVC (Model-View-Controller) design model was created, reducing UI code duplication and separating the user interface code from the code related to actual computational logic of the application.

The general package of the "code_meggie" superpackage notably includes the "caller" module, which is by far the largest module of the entire application. It contains almost all of the actual data processing code of Meggie, only receiving some assistance from the "epochs" module. The "preprocessing" and "sourceModeling" modules are actually empty, but are suggestions for a future clarification of the codebase and should probably be populated with relevant code moved from the "caller" module. See section 5.6 for the detailed discussion on this subject.

5.1.1 The Meggie data model, directory structure and UI state

The data model, i.e. the system used to handle the data read and written by Meggie, is based on the standard tree-like hierarchy of directories in the underlying filesystem. The hierarchy, taking into account the directories needed by the source analysis functionality, is the following:

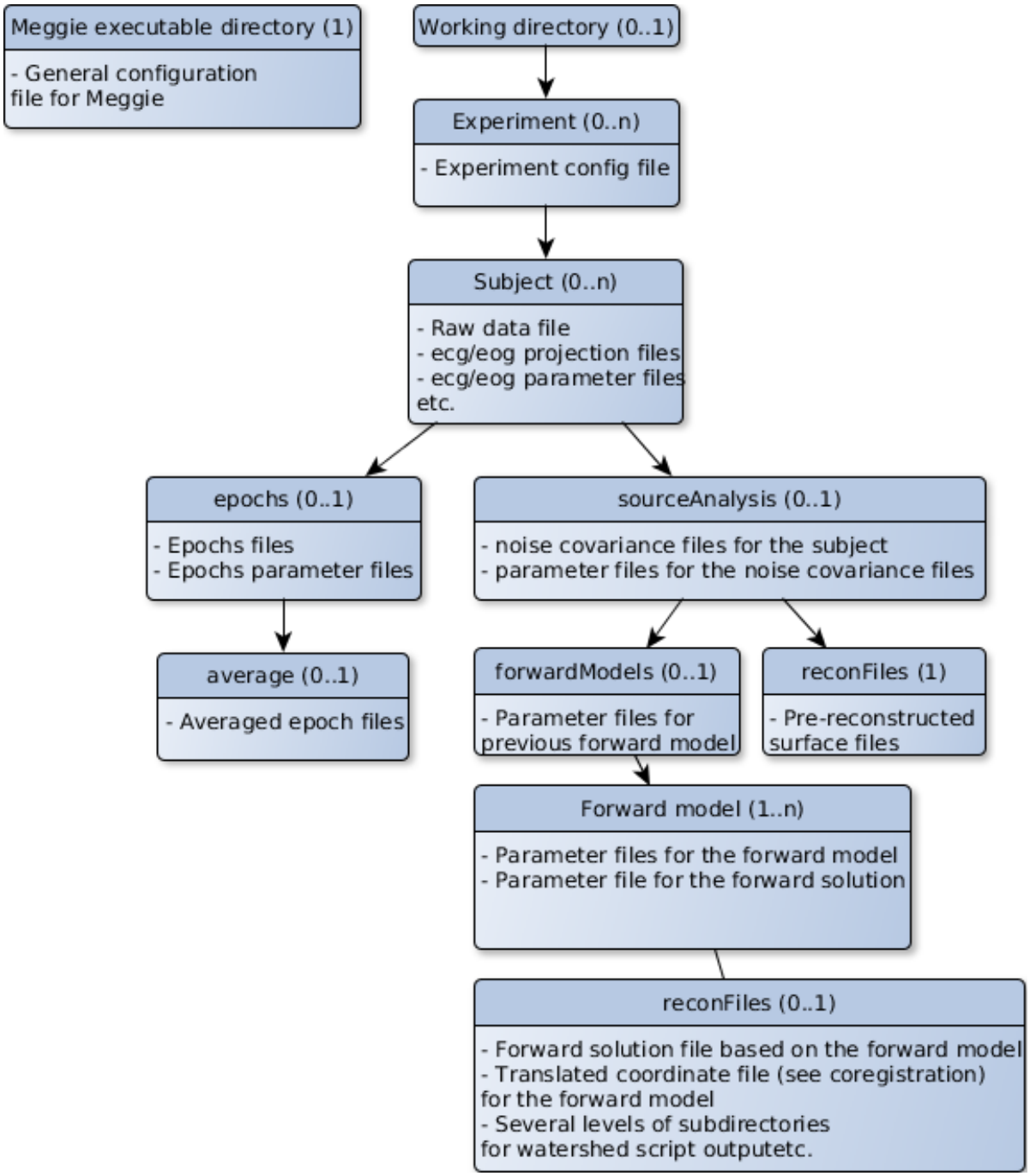

Figure 5. Meggie directory structure

The numbers after the title of the each directory signify the possible number of directories per its parent directory. Also note that the titles with the capital letter signify a directory the name of which is supplied by the user, whereas the non-capital-lettered titles refer to directories named by Meggie itself.

The original reason for choosing to use the standard directory system for Meggie over – for example – a relational database, was simplicity: implementing an intricate interface for accessing a database would have been time-consuming, especially due to the existence of the old write-output-to-disk-style MNE scripts.

Besides, there seemed to be no immediate need for queryable data, as the start of the pipeline consists of broad data conversions that do not yet produce data that needs to be thoroughly examined and analyzed. Later, such data (averaged epochs, for example) was produced, but there still seemed to be no immediate need to change the data organization model to something relational.

Upholding the state of the user interface was mainly based on the existence of the directories and files generated by MNE methods and scripts - with an exception of the Meggie-created files for saving epoch creation parameters. The MainWindow class had a significant amount of code for calling methods that check whether a directory or a file exists and for updating the user interface accordingly. The same held true for various dialogs. This was practical in a sense that is avoids a separate system for storing the user interface state; yet unpractical because, often, all of these checks were run when something changes within the Meggie directory tree. This did not significantly slow down the operation, but refactoring for a more fine-grained user interface updating was speculated to be wise in the future – along with a more event- and MVC-based user interface in general. See section 5.6 for and example about using the Qt MVC system for upholding UI element states related to source modeling.

As the figure 5 shows, the source modeling chain as implemented during the thesis work is still mostly following the same scheme. With MNE-Python methods the scheme was quite usable, as many of the methods allow specifying their input and output files and directories simply via parameters, making it easy to read and write the files wherever needed. On the other hand, working with the older MNE-C scripts was trickier, as they have very specific

hardcoded requirements about directory structures and environment variables.

It should be noted that most MNE-Python methods also return the output data as simple Python objects, often lists or dictionaries. It would not be too troublesome to input at least metadata of these in a some kind of database, even if the actual data would be stored in a filesystem. Considering the fact the old MNE-C scripts are quickly being converted to Python methods, converting Meggie to work with a decent database backend might be very feasible. The benefits and eventual feasibility of this conversion are out of scope of the this thesis, however.

5.2 Implementation of New General System Features

"There should be a proper system for editing and saving various preferences"

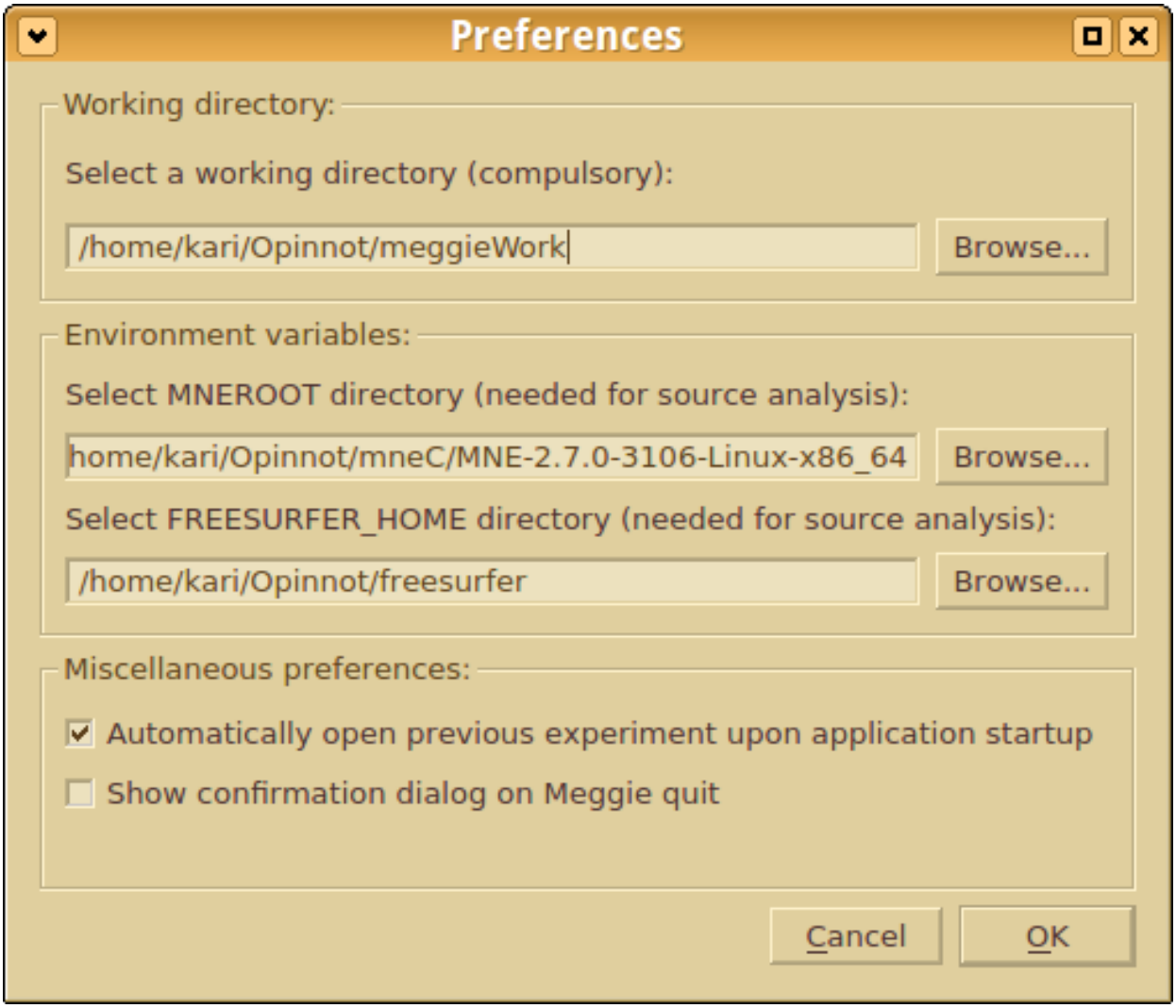

Figure 6. Prefences dialog

Previously, Meggie only included a dialog for setting the working directory of the application. During the thesis, the dialog was extended to also include the user interface elements for all the other preferences that need to persist when the application is shut down. These preferences notably include paths to external libraries required by source modeling, see discussion for the next requirement.

As for the backend implementation, a new PreferencesHandler class was created to take care of storing the preferences runtime, so that the the other parts of the application have a single location whence to get them. The PreferencesHandler class also takes care of reading the preferences from disk at the application startup and writing them to disk when they are changed via the preferences dialog. For reading and writing the PreferencesHandler uses the Python ConfigParser module, thus avoiding home made configuration parsing solutions.

The PreferencesHandler reads the preferences from disk when it is instantiated. This, in turn, happens at the application startup when the MainWindow is instantiated and sets the PreferencesHandler as its attribute (see diagram 7 for Meggie startup). There can, therefore, be only one instance of PreferencesHandler active at a time. This could, in theory, be a typical situation to apply the singleton design pattern (see for example Erich et al. 1995, 127-134), as has been done earlier with the Caller class, to avoid multiple instances of class where only one is needed. The PreferencesHandler is, however, only called by very few other modules or classes, as the MNE source analysis methods mostly employ the environment variables provided by the operating system - thus avoiding most situations where the PreferencesHandler would ever be instantiated. These variables are set by the PreferencesHandler, though, see below.

In addition to preferences altered via the dialog, the PreferencesHandler also stores the name of the currently open experiment. This is to allow the automatic opening of the previous experiment at Meggie startup, as allowed by the corresponding preference in the miscellaneous preferences section. For the automatic experiment loading, also see below.

"Meggie should automatically set the environment variables needed by the external dependencies" AND "It should be possible to automatically open previously used experiment when starting Meggie"

These two requirements are so interrelated they're best described together. Firstly, there are the various environment variables needed by the MNE-C scripts and FreeSurfer[1], and the method to set them is located, again, in the PreferencesHandler, and called at the application startup. Secondly, there is the automatic opening of the previously opened experiment, the implementation of which is more complicated.

A possible challenge with the environment variables is to avoid making them global, i.e. ensuring that the variables set by Meggie only affects Meggie and not everything else that is started in the system - conversely, environment variables set from elsewhere should not affect Meggie. This is important because a user might be using the MNE-C scripts and FreeSurfer utilities manually from the command line while using Meggie, and a mixup on variables would quickly cause errors in both the scripts and Meggie. Fortunately, as found via experimentation, the employed environ method from the Python os module only sets the variables for the process its called from (i.e. Meggie)) and its children, whereas the shell from which the user starts the manual scripts also confines the variables to its children. This ensures the required interoperability.

As for the automatic experiment loading, the loading of the previous experiment at the startup was quite straightforward to implement, but the saving of the identifying information of the open experiment required making a choice: either save the information when an experiment is succesfully created or loaded, or only save it when the application is closed. The former option was chosen to preserve the information even in a case of unclean shutdown, even though it was harder to implement.

The Meggie startup, including the variable setting is illustrated by the sequence diagram below. The stars followed by numbers in the figure (e.g. *1) refer to notes after the figure.

1. see http://martinos.org/mne/stable/getting_started.html and https://surfer.nmr.mgh.harvard.edu/fswiki/LinuxInstall

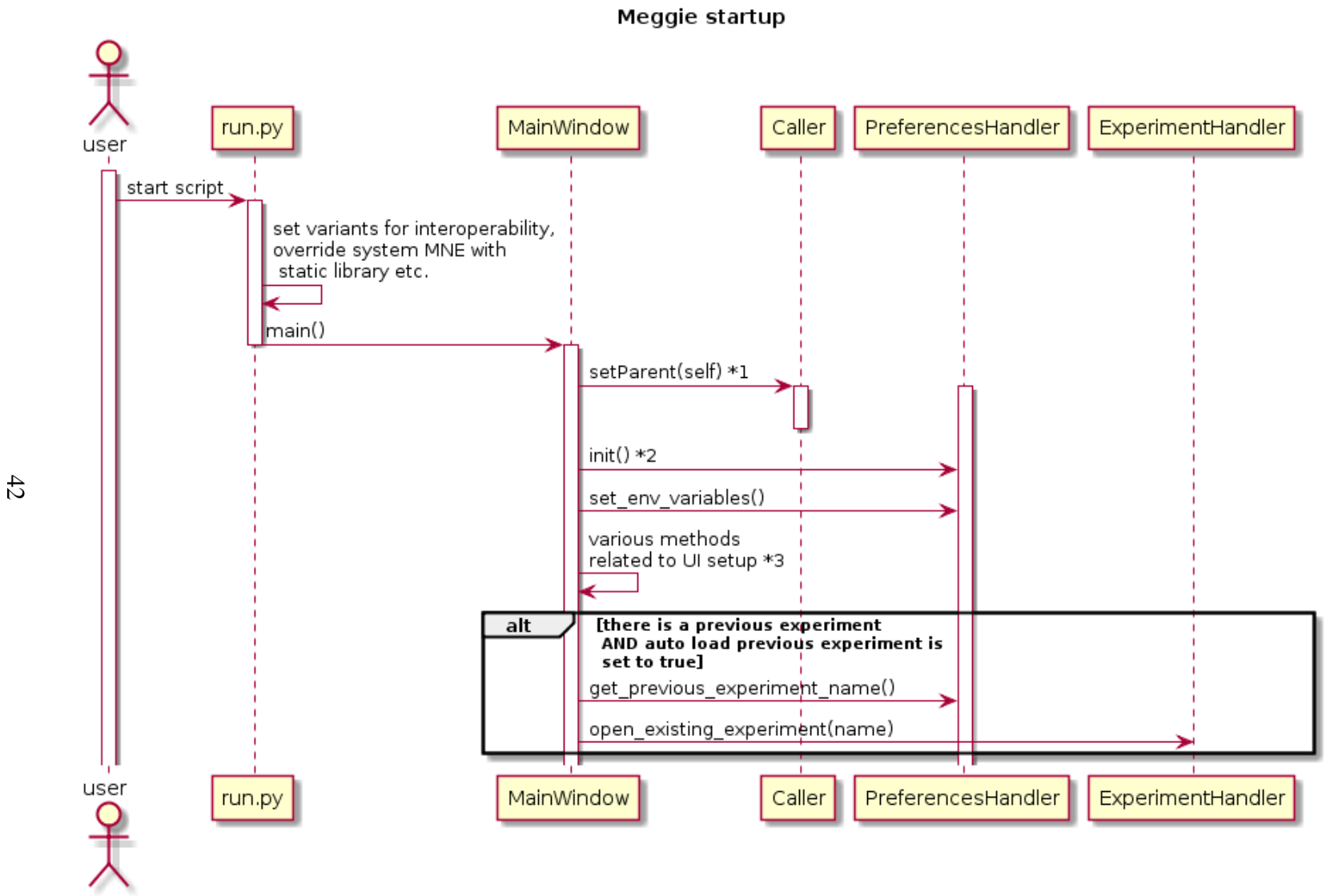

Figure 7. Meggie startup

Notes:

1. The methods at the Caller class get the MainWindow by getting the parent of the Caller. This would, again, be a good candidate to use singleton pattern, considering that the MainWindow inherits the Qt MainWindow class, and there is only supposed to be a single instance of it per application (T. Qt Company 2016).
2. Init method of the PreferencesHandler includes loading stored settings from disk.
3. Lots of user interface element setup omitted, such as connecting models, views and proxymodel, and connecting of Qt signals and slots.

The detailed sequence for the previous experiment (calling experimentHandler.open_existing_experiment() in the figure above) at startup goes as follows:

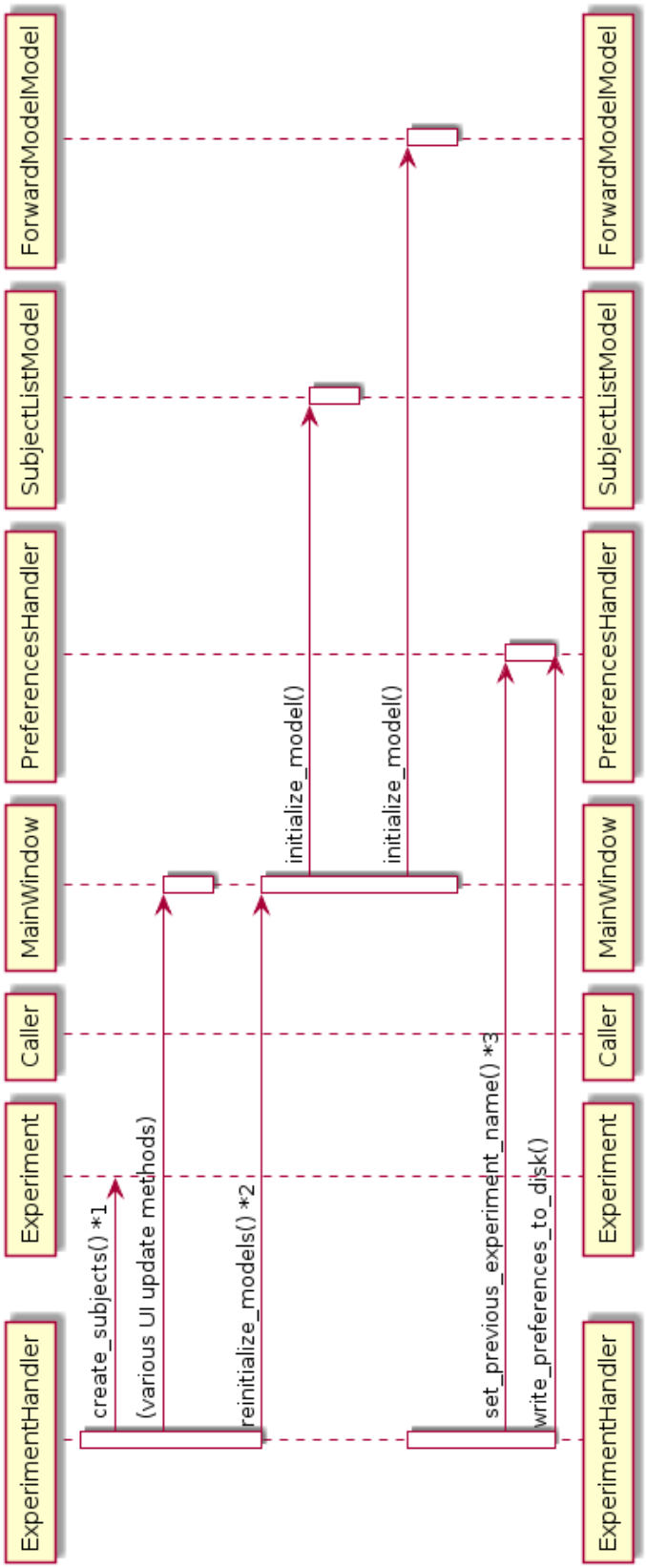

Figure 8. Loading previous experiment at startup

1. This creates the Subject objects based on data on disk and sets them to list in experiment.
2. This method builds the models required by the Qt MVC model views and is needed because of the Meggie "UI state is based on existence of data files on disk" design choice (see 5.1.1). In the case of subjectModelModel, the building only requires parsing the subject directory names - building the forwardModelModel is more complicated, see page 60.
3. This method is not actually needed at startup (as the preferences on disk have not changed). However, the open_existing_experiment method is also used when the user wants to open any previously analyzed experiment, in which case the identifying information for the newly opened experiment would have to be written to disk for persistence and automatic opening at startup.

5.3 Implementation of New System Features Related To Source Modeling

A simplified example of an MEG analysis process is presented by figure 1. A detailed diagram of the source modeling process in Meggie, with actual called MNE-Python methods and MNE-C scripts called, can see seen in figure 2 on page 12. The process steps marked in red had only preliminary UI created for them during the thesis work, but no actual backend implementation.

Also, the version of MNE-Python used during the development was 0.6 and the MNE-C scripts version was MNE-2.7.0-3106-Linux-x86_64. The FreeSurfer version was Freesurfer-x86_64-unknown-linux-gnu-stable5-20130513. None of these were actually specified in the requirements, but these were the versions used when the development started, and the possible version updates were left until the end of the thesis work, instead of trying to keep up with version updates (and cope with the likely breakage caused by them) during the thesis work.

5.3.1 Implementation of Source Modeling Preparation Feature

The main window user interface for all of the requirements related to source modeling preparation (i.e. the contents of the tab) is the following:

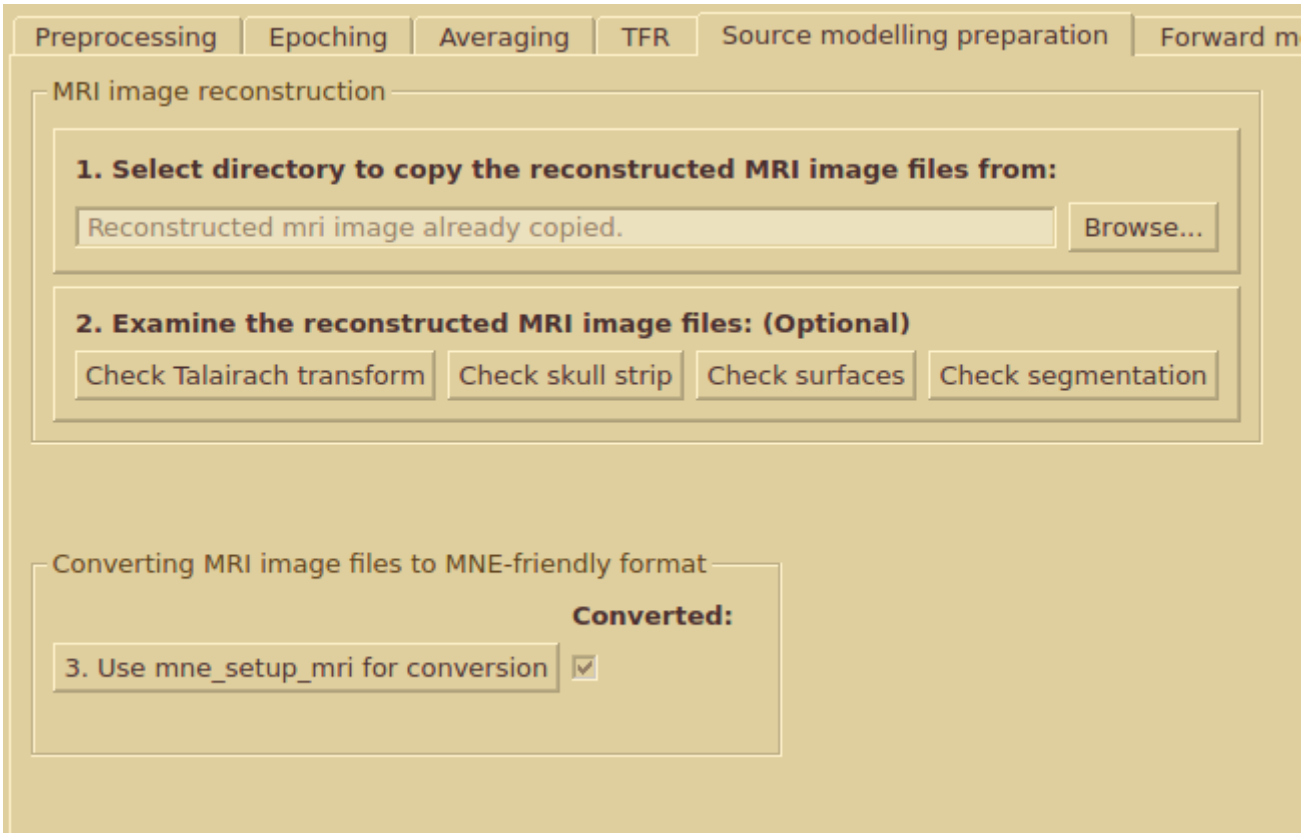

Figure 9. Source modeling preparation UI

The user interface stacks the elements from top to bottom, in the order that corresponds to the required order of the processing steps. To make the order perfectly clear, the elements also include numbers. To further guide the user, the buttons only become active and clickable once the previous steps required by the functionality have been performed. In this case, the required step for both steps 2 and 3 is step 1 - though the buttons for step 2 will stay inactive nonetheless due to unimplemented features, see below.

"It should be possible to import FreeSurfer-reconstructed cortical surface files into Meggie" AND "There should be a warning and a confirmation dialog for the re-import" AND "Re-importing the cortical surface files should invalidate the later source modeling steps for the subject file in question"

These first two of these requirements were implemented, whereas the last one, the invalidation of the later steps, was not. The invalidation would require deleting or renaming the directories generated during the source analysis process and is, therefore, best left until after the full source analysis chain is implemented. An informative dialog is, however, shown to the user if there are already copied reconstructed image files, stating that the results of the later steps are not valid for the new files.

As for the import functionality, the process starts by clicking the browse button next to the topmost textfield in figure 9. The button opens a file browser which lets the user navigate, with the standard Qt file browser, to the directory of the reconstructed MRI images files (or "recon files" for short in Meggie). If the user tries to open a directory without the required files, the Meggie shows a dialog explaining the requirements - currently, Meggie only checks if the user-supplied directory contains "mri" and "surf" subdirectories and assumes that they contain the needed files. If the check passes, Meggie copies the directories under the directory structure of the subject in question and into the directory named "reconFiles". (See figure 5 on page 36 for directory structure). The original recon files are left untouched.

If the recon files have already been copied, there is a text in the topmost textfield stating this fact, and clicking the browse button results into the aforementioned warning dialog being shown to the user.

The sequence for the import process goes as follows:

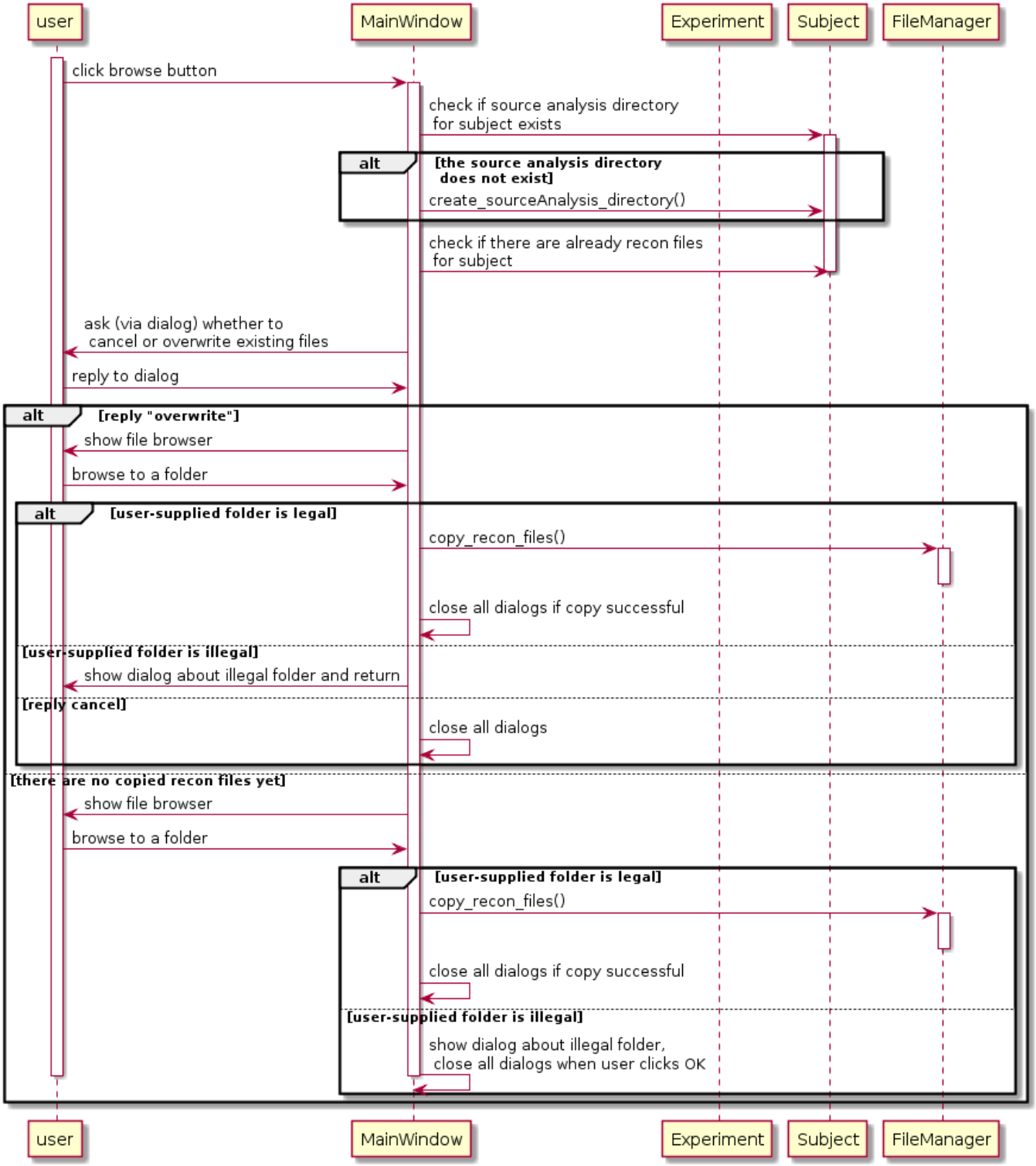

Figure 10. Reconstructed mri image files import

"It should be possible to examine to imported cortical surfaces files for Talairach transform, skull trip, surfaces and segmentation"

The second section of the page includes the buttons to launch FreeSurfer utilities to examine the recon files. The actual launching of the utilities was not implemented, but would be implementable without too much work - this estimate is based on the fact that FreeSurfer scripts are executable shell scripts like the MNE-C scripts are, and there are plenty of examples about executing MNE-C scripts in Meggie. There is one caveat, though: at the time of the UI implementation, the FreeSurfer scripts were calling the C shell (/bin/tcsh) instead of Unix shell (/bin/sh), which in case of the development system was a Bash-compliant shell called Dash (Ubuntu Community 2016) - a non-default shell needs to be specified as a dependency in documentation in case it isn't installed.

"It should be possible to trigger the mne_setup_mri script to convert the surface files into MNE-friendly format"

The bottommost button of the user interface triggers the mne_setup_mri script. Running the script sounds simple, but actually highlights the most challenging problem in the source space analysis implementation: integrating the old C scripts with Meggie. In this case, the problem and the solution was twofold:

- The mne_setup_mri requires two environment variables referring to directories. The first one is SUBJECT_DIR, which refers to a directory which in turn needs to include a subdirectory referred to by a SUBJECT variable. The mne_setup_mri script then searches for reconstructed image files from mri directory under the SUBJECT directory. The solution: set up environment variables so that the former refers to source analysis directory of the subject and the latter to the recon files directory within, which the script searches for "mri" and "surf" directories and their contents.
- There should be way to keep up with the status of the the external command line script and to reliably terminate it and clean up the resulting files if need be. This was not much of a problem with mne_setup_mri, which performs it duties in a few seconds, but it was a bigger one with the more long-running forward model creation scripts (see next section). The status can be obtained quite handily with the Python subprocess

module. The latter problem was not actually solved during the thesis work, and in the specific case mne_setup_mri, the script itself also does not perform any cleanup in case of error.

5.3.2 Implementation of Forward Model Creation Feature

The main window user interface for all of the requirements is the following:

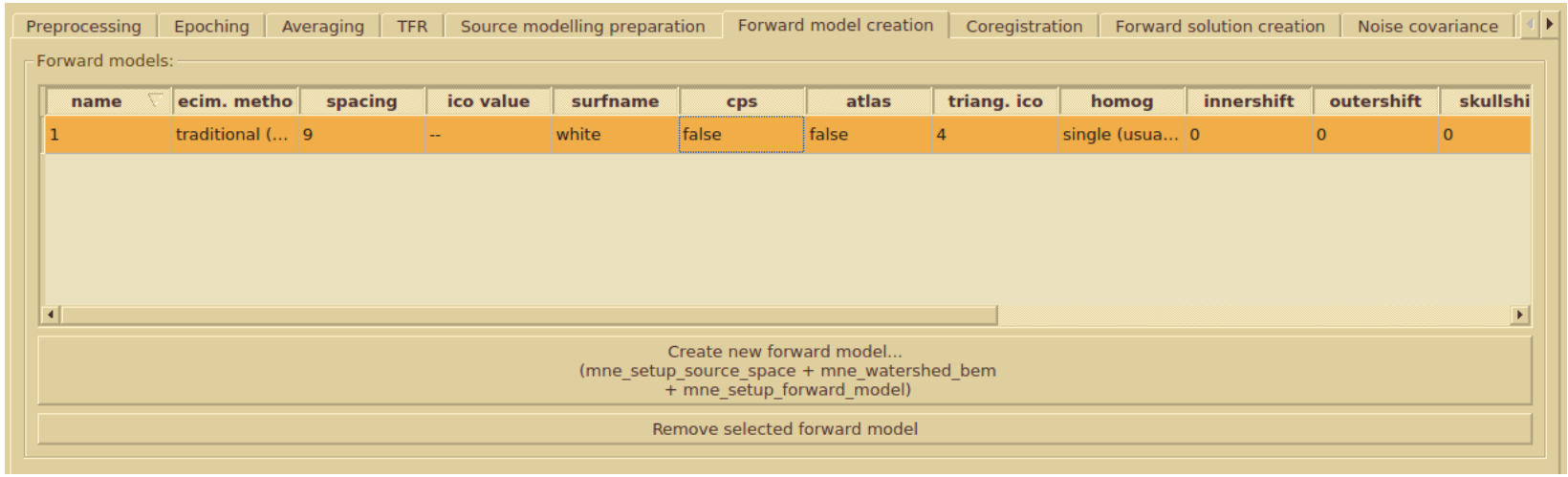

Figure 11. Forward model creation main UI

The table view element at the top of the window shows the created forward models and their parameters, whereas the box at the bottom includes the option to generate the forward model files and to remove the selected model. To make things clear for the users that are accustomed to running the scripts via command line, the "Create new forward model..." button also includes the actual script names to be run. The button only becomes active and clickable if the steps in the previous tab have been completed. The "Remove selected forward model" button also only becomes active and clickable when a line representing a forward model is selected in the table.

The parameter input dialog that is opened via the "Create new forward model..."-button is the following:

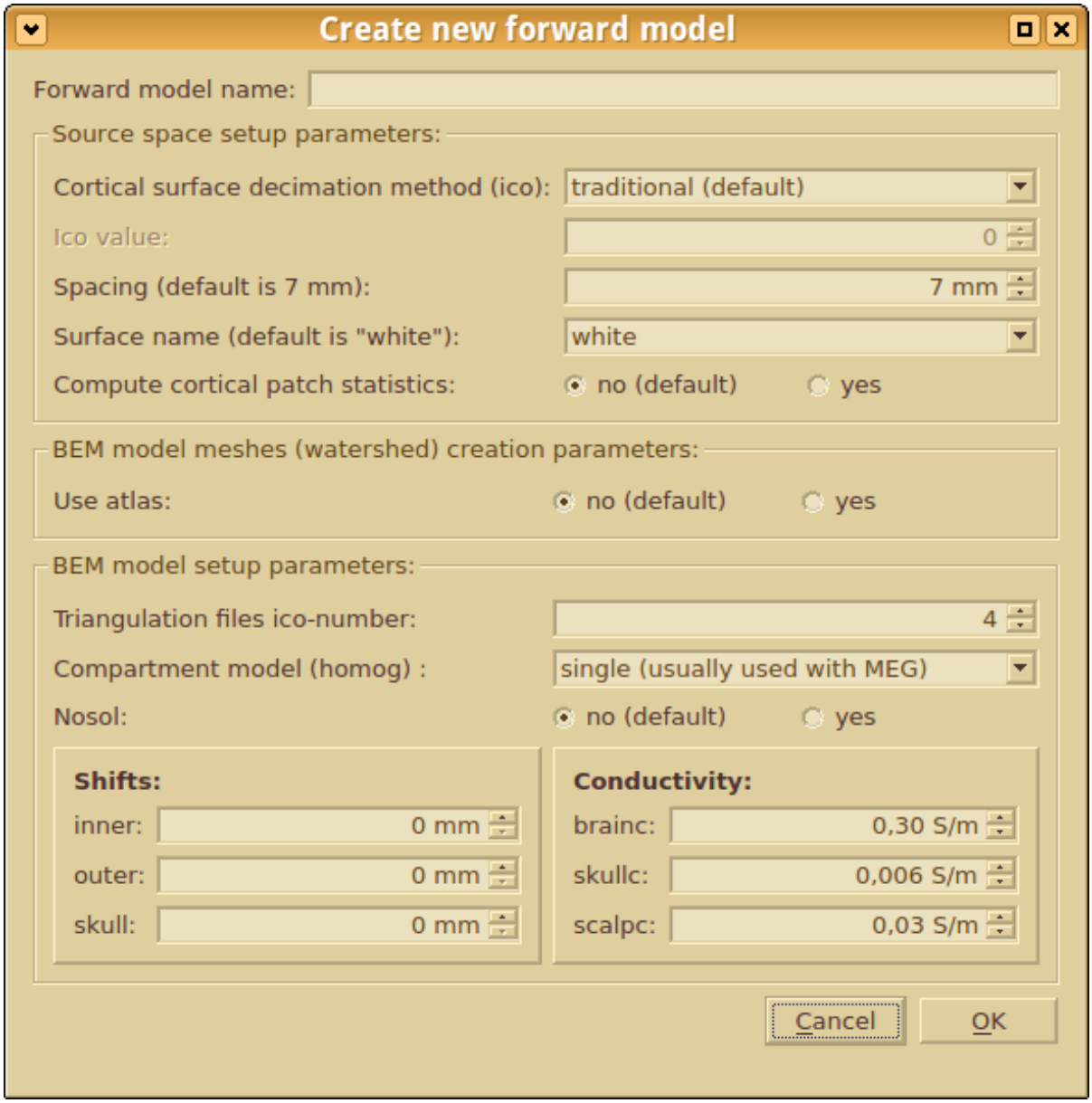

Figure 12. Create forward model dialog

The dialog is quite straightforward, including the input field for the forward model name, below which are located the parameter input fields (grouped visually in boxes) for each of the three scripts needed for the forward model creation. The labels for the elements state the default values for most of the fields, in case the user forgets them while fiddling with the values (which are set to default when the dialog is opened).

The dialog also has some dynamic elements. Depending on the cortical surface decimation method chosen, the "Ico value" and "Spacing" fields get activated or deactivated. Also, the surface name dropbox is populated by crawling the "surf" directory (under the recon directory) for surface file names - this removes the need to input the surface name manually and is also useful due to the fact that the directory may not always include all of the possible surface files the names of which the user may think to input.

Finally, the dialog has input sanity checking for the forward model name (should not match with an existing forward model name or MNE directory names such as "mri") and ico parameter value (should be non-zero if decimation is not "traditional"). These checks are triggered when the OK button is clicked, and the problems encountered by the checks are shown in a modal dialog.

The dialog only shows the first error encountered - arguably, the dialog should list all the encountered errors at the same time to allow the user to correct everything in one step. Another possible solution would have been to use field-specific errors, e.g. set the UI to trigger and show an alarm whenever an user has edited a field and tries to move to the next one (as per principle of immediate feedback, see for example Jakob Nielsen 1993, 134) - or possibly show the alarms only when clicking the OK button, but instead of a separate modal dialog, show the alarms next to input fields in the original dialog.

"Meggie should combine setting up source space, creating bem model meshes (with watershed algorithm) and setting up forward model into a single analysis step" AND "It should be possible to skip first two parts of the forward model creation by using previously created results for them" AND "Meggie should be able to create and store several forward models created with different parameters" AND "The forward model creation parameters should be saved and readable for each model"

As can be seen from the user interface figure above and the figure 14 below, these requirements were fulfilled. However, together they also comprised the single most challenging system requirement in all of the thesis work, combining the problems with C-script integration to control flow and program state integrity issues and usability challenges. The following sequence diagram illustrates the process:

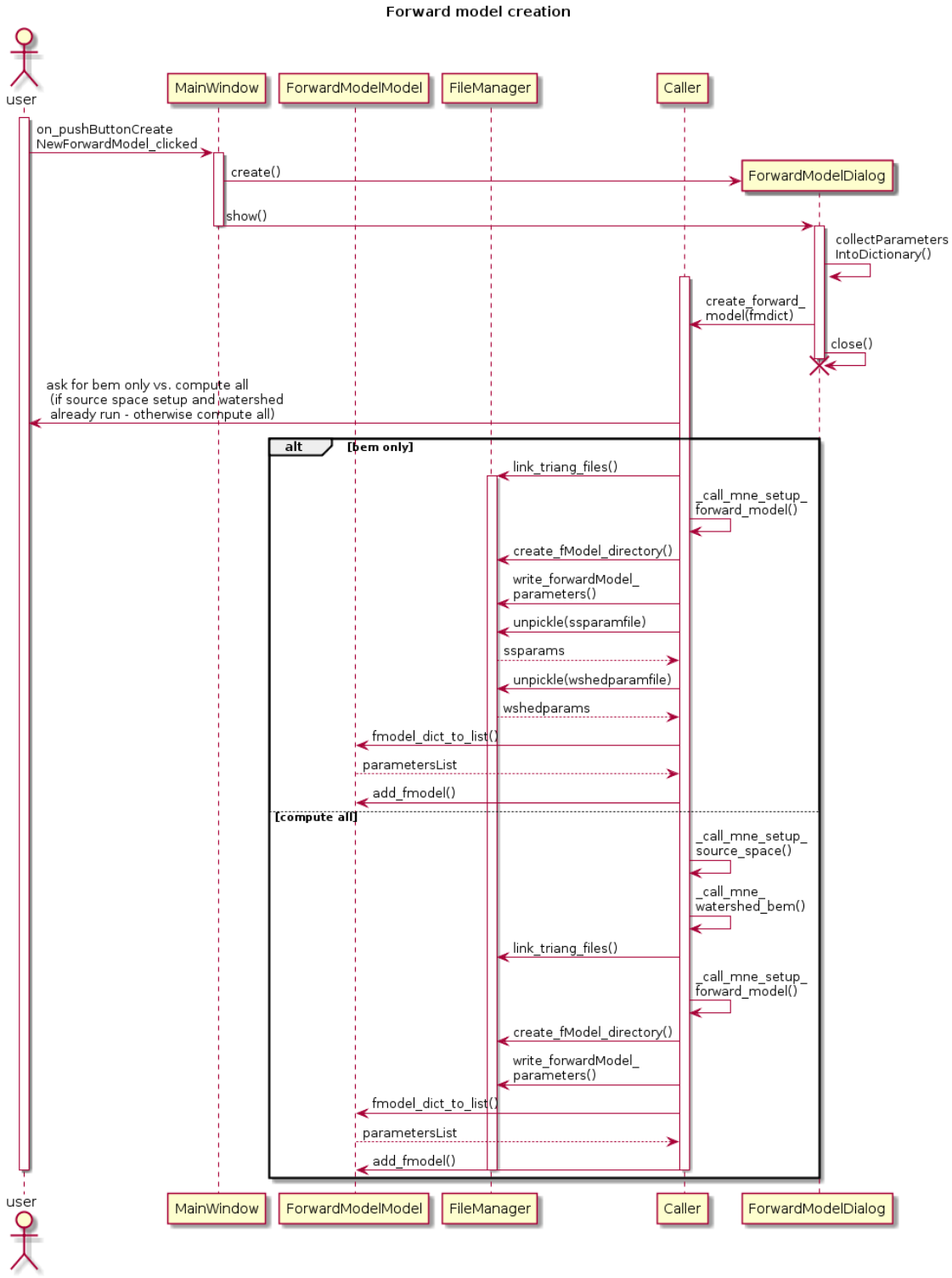

Figure 13. Forward model creation

1. All parameter input dialogs (also in Meggie in general) relay their input parameters to backend code (in this and almost all other cases the Caller class methods) via dictionaries.
2. mne_watershed_bem creates surface triangulation files in its output directory (the "watershed" directory). MNE forward model creation and coregistration methods require symbolic links to these files in the "bem" directory. This method takes care of the creation of these links.
3. Creates a directory corresponding to the new forward model, copies the whole bem directory (containing the final forward model files) to this directory, makes symbolic links to subjects mri- and surf-directories to avoid copying them around (they are needed in later steps), and copies parameter files used to create the forward model to the directory (while keeping the originals in place should they be needed again). The last aspect of the method is mainly used when the first two phases of the forward model creation have been run before and the user chooses the "Bem model setup only" option in the dialog (see figure 14) that is, in this case, shown after clicking the OK button. In the "Compute all phases again" case, the write_forward_model_parameters method takes care of the parameter file copying.
4. Unpickle is the standard Python deserialization method. In this case, it deserializes the parameter dictionaries used before for running setup_source_space and mne_watershed_bem scripts. The very same dictionaries are used to show the parameters in the dialog of figure 14.
5. The ForwardModelModel is the model part of the Qt MVC pattern and needs to be updated for a new line to show up in the corresponding user interface element (the table at the top of figure 11). The fmodel_dict_to_list method earlier in the sequence is needed due to the fact that the Qt's AbstractTableModel only accepts lists for storing its data entities.

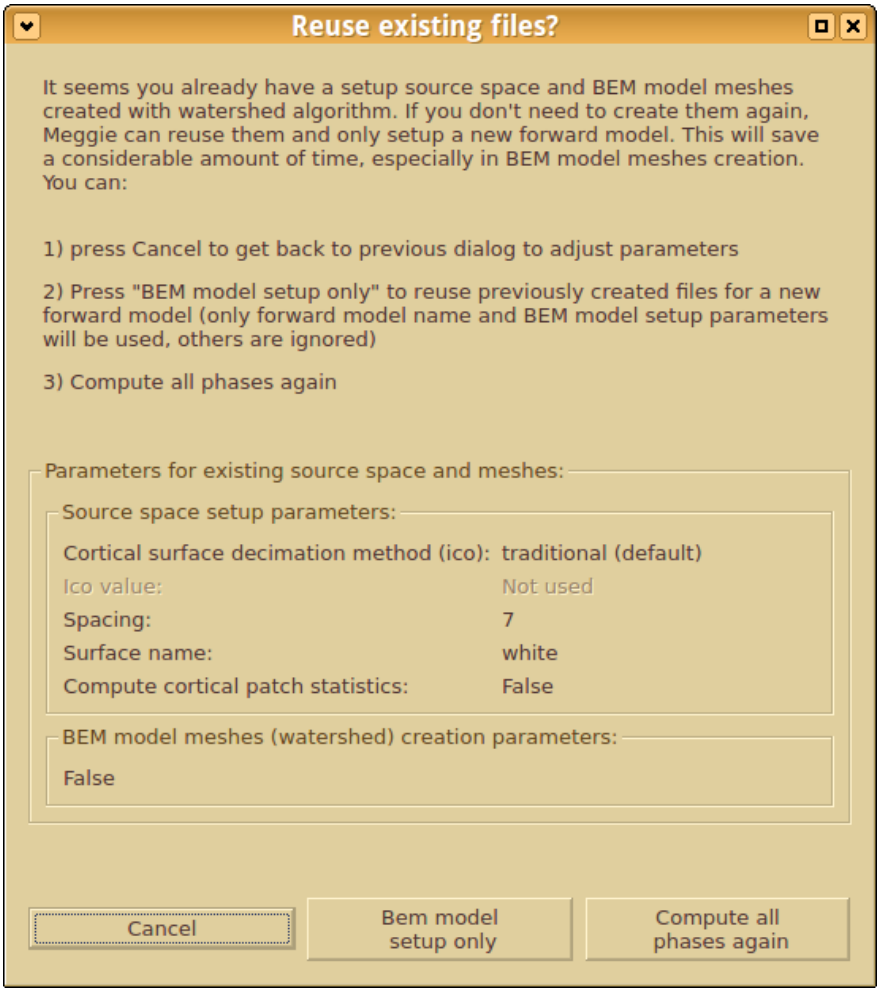

Figure 14. Reuse existing files for forward model creation dialog

"It should be possible to remove previously created forward models from the Meggie data structure"

The forward model files and their parameters files are, as stated above, stored in a single directory and its subdirectories. Removing a forward model is therefore very easy: simply remove the directory and the corresponding item from the ForwardModelModel instance. To prevent the user interface and disk states going out of sync in case of error, i.e. the user interface showing there being a forward model event though there isn't, the directory removal is tried first and the ForwardModelModel object is edited only after it has succeeded.

About the ForwardModelModel class

The ForwardModelModel class could only have been used as the MVC model class for forward model data. It could also only have corresponded to the table view in the forward model creation tab. It turned out, however, that it can also be used as a model for views in the coregistration and forward solution creation tabs (see below). Both the coregistration and forward solution steps are performed on a single forward model, and the Qt ProxyModel allows for filtering the base model data to form new models for different views. The added bonus of using the ProxyModel classes is the ability to sort the lines in table elements based on e.g. forward model name.

The ForwardModelModel therefore ended up storing not only the parameters the forward model was created with, but also whether the coregistration has been performed on it and whether there is a forward solution based on it. This made the user interface related code surprisingly simple, considering the alternative might have been separate models for all of the three steps. Considering the breadth of its functionality, though, the name of the ForwardModelModel class may be a bit misleading.

More on intricacies and problems of the MVC implementation in Meggie in the section 5.6.

5.3.3 Implementation of Coregistration Feature

"It should be possible to use mne.gui.coregistration from MNE-Python for coregistration"

Mne.gui.coregistration is a graphical application for performing the coregistration, i.e. defining the boundary-element surfaces, the source space and the MEG sensor locations in a common coordinate system (see section 2.2.4). It is part of the MNE-Python distribution. It can be given, as parameters, the raw data file path of the subject, the name of the mri subject and the subjects directory where this subject can be found. To integrate the application with Meggie, the parameters were set to point to appropriate files and directories in the Meggie directory structure.

The problem with mne.gui.coregistration is that the location of the resulting translated coordinate file cannot be specified as parameter, but must be chosen by the user at the end of the coregistration process. There is a default location for this file: to move it to an appropriate location for Meggie, the user is shown a following dialog when mne.gui.coregistration is launched. The dialog is the following:

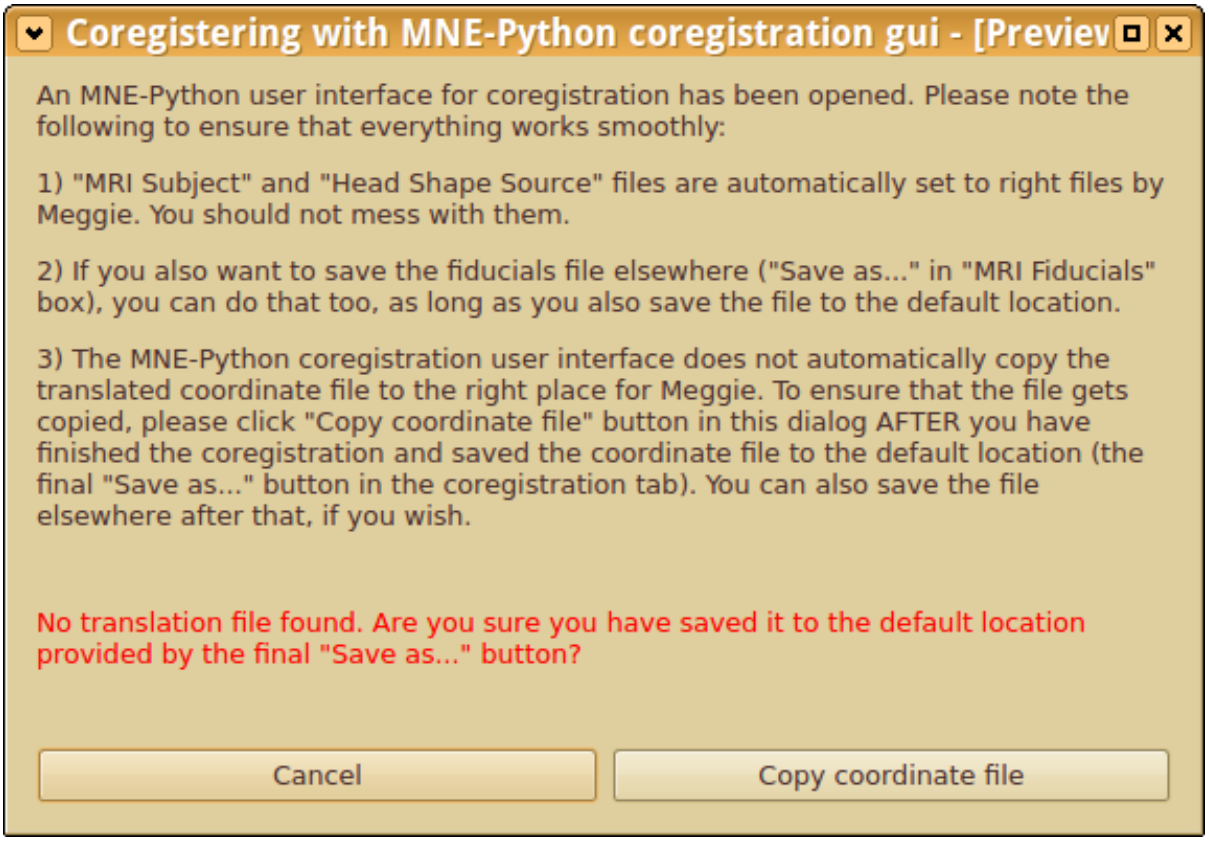

Figure 15. Helper dialog for coregistration

The warning (in red) is only shown if the user clicks on the "Copy coordinate file" button and Meggie cannot find the translated coordinate file to copy to the Meggie-appropriate location. If it can, the coordinate file is copied and the code in the dialog also updates ForwardModelModel to reflect changes.

"There should be an option to import a pre-existing coordinate file into Meggie"

This was rather straightforward to implement: the user is shown a standard file browser with instructions to choose an existing coordinate file. If the file is deemed valid (based on the filename), it is subsequently copied to the same location as by the code triggered by the helper dialog for coregistration.

"Meggie should be able to create separate coordinate files for each forward model"

As the figure 5 suggests, the coordinate file is copied under the directory corresponding to the forward model it is based on. This makes it possible to have separate coordinate files quite easily.

"It should be possible to re-perform coregistration on an already coregistered forward model, overwriting the previous coordinate data"

This was also very simple to implement by simply overwriting the existing coordinate file. It was also not deemed worthwhile to save the parameters of the coregistration: the parameters used for the creation of the coregistration file are apparently not saved in the file, and saving them would require separate parameters files and possibly a customized version of mne.gui.coregistration to save the parameters into them.

5.3.4 Implementation of Forward Solution Feature

"Meggie should be able to create a forward solution for all forward models in the Meggie data structure" AND "The forward solution creation parameters should be saved and readable for each solution" AND "Re-creation of a forward solution should be possible, and in these cases the previous solution does not need to be retained"

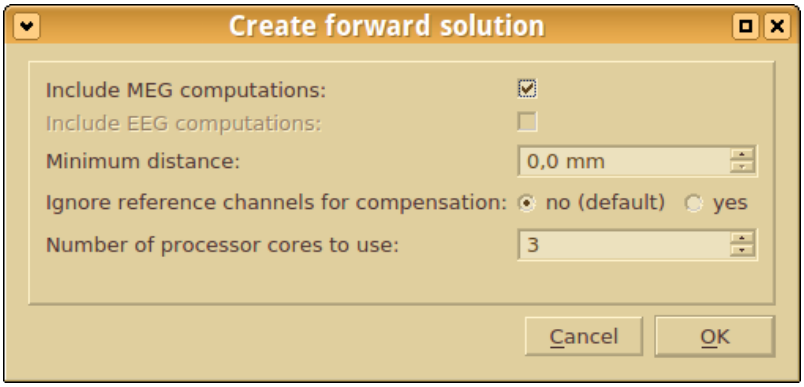

Figure 16. Create forward solution dialog

Creating a forward solution requires input of few parameters, as forward solution creation is mostly based on data files created in the previous steps, i.e. the forward model creation and coregistration. The table in the forward solution tab, therefore, only shows the coregistered forward models as eligible sources for the forward solution. The required parameters are minimum distance of sources from inner skull surface and whether to ignore the reference channels for head position compensation [2]. The MNE method for forward solution creation is also multi-threaded, so the dialog also asks for number of processor cores to use (and defaults to number of cores in the computer minus one).

The parameter file saving was implemented in a way identical to forward model parameter saving.

As for the recreation of the forward solution, the implementation was, again, simple: overwrite both the forward solution file (created by mne.make_forward_solution) and the parameter file (created by Meggie).

5.3.5 Implementation of Noise Covariance Feature

"It should be possible to create the noise covariance matrix based on a the raw file with Meggie experiment, or an external raw file"

2. the documentation of mne.make_forward_solution method does not link the reference channel ignoring to head position compensation, but source code for the method at MNE Developers 2018a does

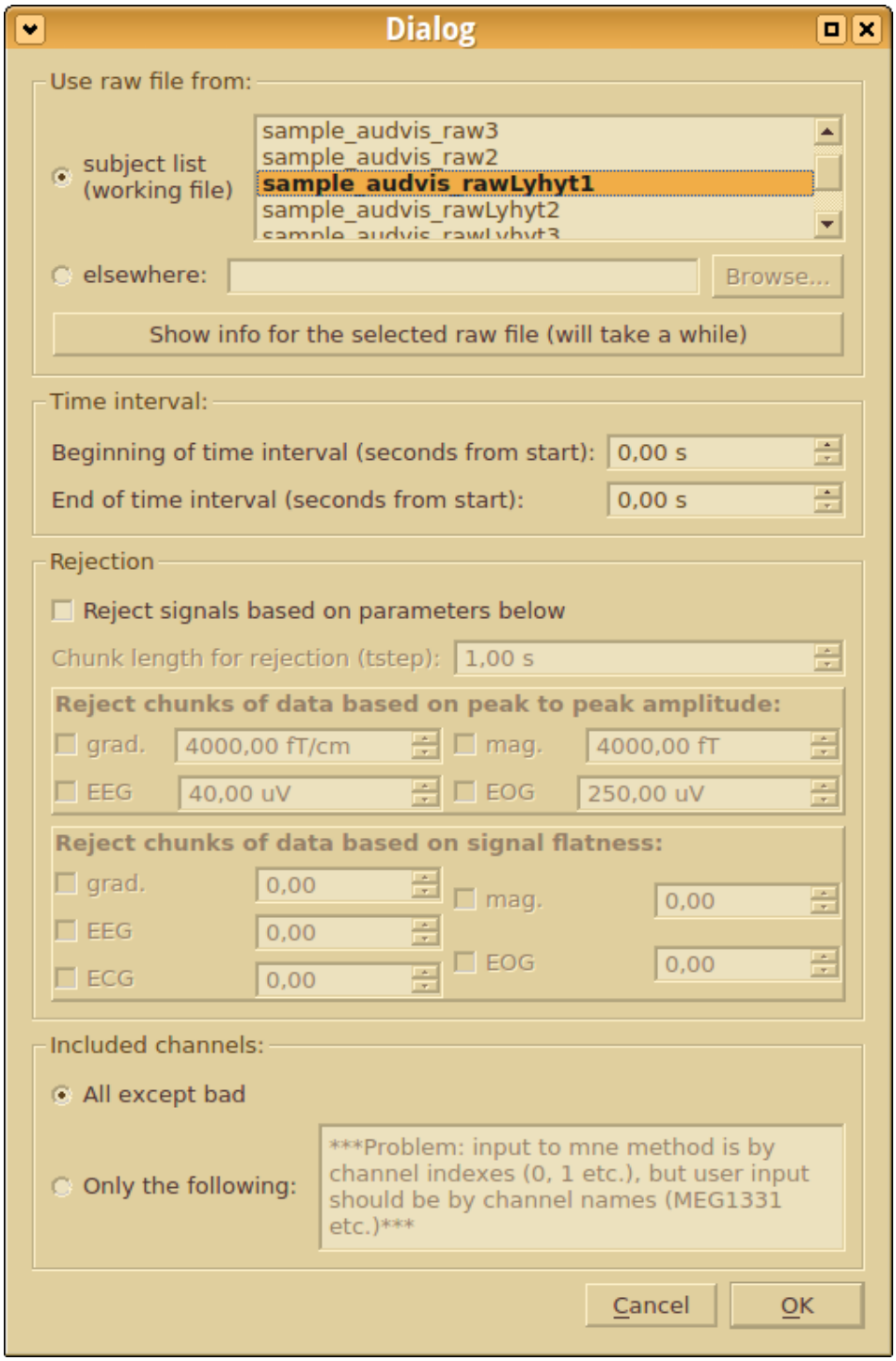

Figure 17. Create noise covariance matrix based on raw file dialog

This requirement was implemented. The other, non-essential, requirement (creating the matrix based on epochs collection) was not.

From the Meggie point of view of Meggie code, the matrix creation is a simple process, in this case using the mne method compute_raw_data_covariance to create the matrix object and the write_cov method to write this object to disk. As the dialog suggests, the first method, however, requires an impressive array of parameters.

Making it possible to choose the raw file from among the existing raw files in the experiment ("subjects") was challenging: the original implementation of the subject list element in the main window was not based on the MVC design model, thus making it cumbersome to use the same list elsewhere. For the implementation of the noise covariance matrix creation, the subject list was also converted to MVC.

The ability to choose any raw file was, on the other hand, simply implemented with a standard file browser. Technically, the user could also use the file browser to choose any raw file in the experiment by browsing to the right file in the Meggie directory structure, making the subject list in the dialog only an element of convenience. Convenience, however, is one of the goals of Meggie, making the small amount of extra work required by this feature warranted.

5.3.6 Implementation of the Other System Features

Due to the limitations of the thesis work, the other system features (inverse operator creation, source estimate creation and source visualization) were left unimplemented. The solutions for the problems found during the work may help in implementing them, though, as the procedures and structures they require are not qualitatively different from those created during the thesis work.

5.4 Implementation of the Features Related To User Interface Requirements

"Adding the user interface components for source analysis capabilities should not make the user interface too confusing"

Before this requirement can be evaluated, we need a definition for "too confusing". For this explorative work, the chosen definition is mostly based on Nielsen's ten heuristics for user interface design, which themselves are a derivation of the Nielsen's earlier heuristics for usability engineering: a confusing user interface is an interface that clearly breaks the guidelines provided by these heuristics. Also, since the requirement is about comparing the usability of the user interface in state A (before the implementation of the source analysis capabilities) to state B (after the implemention of said capabilities), we only need to analyze the usability changes caused by the implementation of source analysis capabilities.

Beginning with the heuristics that were definitely considered during the implementation, arguably the most obvious one was "Recognition rather than recall": Meggie is an application with a graphical user interface, and the functions performed by it can be discovered by exploring the interface. This is in contrast to commands input via command line, which require the user to recall the switches and parameter values. Arguably, adding the source modeling functionality did not actually change the user interface much in regard to this heuristic - this is actually a good thing, as the interface could have taken a turn for the worse, i.e. by adding text input fields which require inputting raw commands. (Jacob Nielsen 1995)

The largest change to the overall user interface of Meggie was the adding of the tabs in the main analysis section of the main window (see figure 3 for main window, and figure 9 for added tabs). Arguably, the added tabs didn't change the general flow of the application, thus following the standard set by the application for itself. As for the Nielsen's heuristics, this aspect of the application would therefore be an example "Consistency and standards" and perhaps also the "Match between system and the real world", if we consider the fact that the order of the tabs follows the typical order of the analysis steps in actual MEG data analysis chain (Jacob Nielsen 1995; Jakob Nielsen 1993, 126-129). Using Nielsen's earlier terms, it could also be stated that the real-world analysis chain is sensibly *mapped* to Meggie user interface (Jakob Nielsen 1993, 126-129).

On a smaller scale, the same match can be seen in the source modeling preparation tab, where the enumeration of the user interface elements and their ordering from to bottom corresponds to actual order of the real-world analysis steps. Also the (de)activation of the elements based on whether the functionality they relate to can actually be performed at a given stage of the

analysis might correspond to "Visibility of the system status" in the Nielsen's heuristics. One alternative to fulfill the spirit of the heuristic could have been to keep all the buttons always active and show error messages when inappropriate buttons are clicked - arguably, though, the realized solution fulfills the spirit of the heuristic better, requiring less work from the user to discover the status. As a bonus, the solution also fits the heuristics of "Error prevention" and "Help users recognize, diagnose, and recover from errors" - though the implementation of a global exception handler (see page 75) may help more in avoiding badly reported errors, and input value sanity checking and value limits in dialogs may prevent errors in the first place.

For some parts of the user interface, however, the ordering according to the real-world analysis sequence was not feasible. A specific example of this is the preferences dialog (see page 39): since there is no analysis chain ordering to follow, a principle of visual hierarchy was followed instead. The principle calls for the more important and often used elements to be placed at the top, whereas the less used are to be placed the bottom (MSDN 2010, Gnome Project 2014). Also, the preferences of the same category were grouped visually together by groupbox elements for added clarity, and the dialog also includes clarifications ("Needed for source analysis") for the purpose of the preferences that are not immediately evident.

In addition to ordering of the elements, the question about what elements to show in the first place was considered. Specifically, the ability to hide the subject list is was born out of this consideration: hiding the list allows for more UI space for the analysis of the current subject, if needed, at the cost of making switching between subjects slower. The functionality would therefore cater to scenarios during which the switching is rare, as well as those that require regular switching, with rather negligible UI complexity addition of one button in the menubar. The most fitting Nielsen heuristic for this feature could be "Aesthetic and minimalist design", in that the feature allows concentrating on the relevant information. The design might also fit the Nielsens earlier heuristic of "Simple and natural dialogue", in that it minimizes navigation (scrolling of the tab bar) and allows the user to see information needed and no more (Jakob Nielsen 1993, 115-116).

In addition to positive changes, there are usability heuristics related to requirements that might not be very well taken into account. For example, the "Flexibility and efficiency of

use" is not taken care of in all senses of the heuristic definition, in that there are few shortcuts and little tailoring for frequent actions: the menubar contains shortcuts (in the form of icons) for actions also found in the menus, but otherwise the examples are few. On the other hand, the heuristic stresses the usability for both the experienced and inexperienced user of the UI, in that the expert users may speed up the use with the shortcuts that the nonexpert users are not required to know - Nielsen does not give an example of this, but rather common copy and paste keyboard shortcuts (often referring to actions also found in menus) in many text editors spring to mind. Meggie is not a text editor, however, and it is not evident that Meggie even has such common actions that should have keyboard shortcuts. Meanwhile, a feature that speeds up the use in certain situations the saving of the previously used parameters in the dialogs - this allows fast reruns of steps with slightly altered parameters.

Another neglected heuristic is the "User control and freedom", in that Meggie doesn't allow the user to leave an unwanted state, i.e. to cancel a mistakenly started process while it is running. See discussion about the requirement "Ongoing computations should not make the user interface of Meggie unresponsive" on page 71 below.

The last undiscussed heuristic on the Nielsen's list, "Help and documentation" was not actually a focus of the thesis, unless you consider the thesis itself a form of documentation. The Qt "What's This?" mode (see below) is related, though, and allows for some inbuilt documentation in Meggie.

"There should be a menubar with buttons for most commonly used general functions of Meggie AND Meggie preferences should be found in single location in the user interface"

The menubar was implemented, see figure 3. The bar includes buttons (from left to right) for adding a new subject to the experiment, opening another experiment, showing info about the current experiment, toggling the Qt "What's This?" mode (in which clicking an UI element shows an extended description about the related functionality) and, finally, toggling the subjects sidebar on/off. Arguably, a button for opening the preferences window could be added - not that the preferences dialog would be needed often, but simply because there is plenty of empty space in the menubar and putting it to good use would increase the complexity of

the UI only very marginally.

A menubar element of special noteworthiness is the button for toggling the subjects sidebar on/off. The "off" setting hides the sidebar, freeing up space for the main analysis section, which - as discussed above in at length - is useful when working on a single subject, at least if the there is little horizontal screen space.

As for the preferences being found in the single location in the user interface, the implementation is already detailed in the general system features section ("There should be proper system for editing and saving various preferences").

"Ongoing computations should not make the user interface of Meggie unresponsive"

This requirement was definitely not implemented. All of the source analysis methods block the user interface completely, making it impossible to interact with the application at all. To make the computations non-blocking, solutions would be needed for (at least) the following problems:

1. There would have to be a way to make the computation methods run in separate threads or processes, outside the main UI process. Again, the distinction between MNE-Python methods and old MNE-C scripts may need special attention.
2. There would have to be a way to be make the user aware of the status of each non-blocking process (e.g. still running, finished, finished with errors).
3. There would have to be way to terminate a process and clean up the possibly half-finished files. At least some of the MNE-C scripts were found to perform cleanup in case of some errors, but it should be checked that all of the relevant scripts do so consistently and reliably.

Of course, the UI considerations of non-blocking methods or processes would be significant. For example the UI would need to signal the status of the processes to the users. More importantly, there would have to be limits on what processes could be run simultaneously. For instance, there is not much point running the source modeling preparation step while running some later step, since rerunning the source modeling preparation step invalidates the files used by all the later steps.

5.5 Implementation of the Features Related To Software Interfaces

"The interfaces between MNE, FreeSurfer and Meggie components should be clear and consistent"

This is definitely a requirement related to software architecture, as the interfaces in question are between the code written for Meggie and libraries and scripts that are called by this code. According to Wieger, to describe the interfaces, one should "specify the mappings of input and output data between the systems and any translations that need to be made for the data to get from one system to the other" (Wieger 2013, 197).

Some details of the mappings and translations have been described in sequence diagrams and their comments about system features in section 5.2 and also partly in figure 5 about the directory structure of Meggie. None of these describes, as a whole, the mappings between the three main parts of the system, i.e. the UI level (the source of parameters), the Meggie code and the MNE-Python and MNE-C methods and scripts. The following figure illustrates the mappings, using forward model creation as an example (for other details about the forward model creation, see 13):

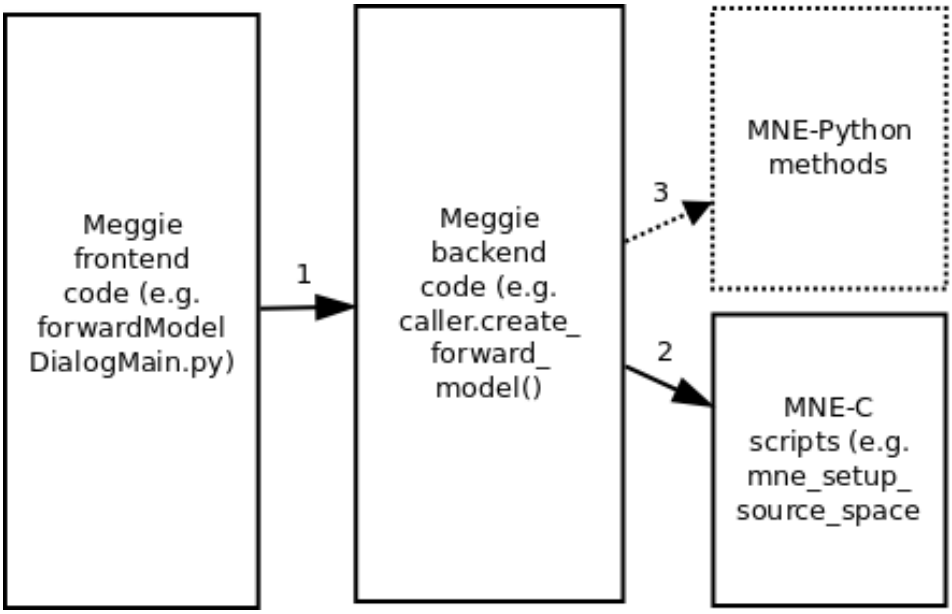

Figure 18. Meggie parameter mappings

1. The code in dialog class collects the values from the UI elements and sets them into a dictionary which is given as a parameter to the Caller method.
2. The code in Caller method calls a Filemanager module method convertFModelParamDictToCmdlineParamTuple to map the parameter dictionaries into form that can be used to call the MNE-C command line scripts via Python subprocess module (mainly adding command line parameter switch characters in front of parameters). Arguably, Filemanager is wrong place for this kind of method, as the other methods in the class write or read files to or from disk - a static module named callerUtilities could be better for these kind of non-I/O methods.
3. MNE-Python methods are not called when creating forward model. A good example of calling them can be found in forward solution creation (forwardSolutionDialog-Main.py -> caller.create_forward_solution). In short, calling MNE-Python is a lot simpler than calling MNE-C scripts, as the non-I/O related parameters in the UI-given dictionary can often be used directly as parameters for methods.

5.6 Implementation of the Features Related To Quality Attributes

"(Essential) Meggie should conform to MVC model or some other model of decoupling the functional code (backend) and the user interface code (frontend), thus making separate development, easier modifiability and greater reusability of UI components possible."

The user interface elements that explicitly use the Qt 4 MVC model implementation are the subject list (in the main window UI and noise covariance dialog) and the lists in forward model creation, coregistration and forward solution tabs. The subject list elements in both UI locations look exactly the same - the only information in the model is the subject name and there is no proxy model to customize what information is shown. As stated in the discussion about ForwardModelModel on page 60, however, the other three lists have the same model, but a different proxy model was used to choose what information is shown in each actual UI element. For details about Qt and PyQt model-view programming, see Summerfield 2007, pages 413-423 and Qt Project 2018.

The actual setup of the models was realized as described in the sources above and was quite simple. Initializing and updating the model used for the subject list was also quite straigth-forward, as the only piece of information needed by the model was the names of the subjects, which could be directly extracted from the subject paths stored in the experiment, which in turn could be reached via Caller. For the ForwardModelModel, the process was a bit more complicated, as the information about the forward models and forward solutions needed to be collected from the parameter files to initialize the model.

Regarding the update of the models, there was a choice to make. The models could have been updated via the Qt signals-and-slots system, i.e. making the Caller emit a signal about a created forward model or solution or a performed coregistration (with parameters), and associating a slot with the model to catch the signal, making the model update itself. For this exploratory work, explicit method calling was deemed sufficient, as the ForwardModelModel was the only location in the code that needed this information. Besides, the system is designed for communication between UI elements (of "widgets", as is the Qt term) and using it in this case might would have been a breach of the Qt coding standards. (t. Qt Company 2016)

Also of note is the fact that the MVC solutions were only used for the functionality required by source modeling. There were data structures related to epoching and averaging would definitely have benefited from similar solutions - instead of resorting to an ugly hack for moving a single epoch list UI element around the tabs - but refactoring them was out of scope for the thesis work.

"Meggie error and exception handling should follow the Python EAFP (Easier to Ask Forgiveness than Permission) coding style recommendation - or some other style that allows for decent error reporting to the user while keeping the error reporting code out other program code (see below)." AND "Error reporting code (e.g. creating the actual dialogs) in the backend should be avoided and relegated to the frontend. This is related to the MVC model and EAFP requirements above."

These were partially implemented in source modeling code. Most calls to Caller by the UI classes were encased in try-except statements, with an error message dialog shown if an

exception is caught. This message includes the actual error information (with stack trace) to ease debugging and sending relevant information in possible support request cases. It would also have been useful to add, to the error message, the parameters used to call the method that threw the exception - this was not implemented, however. Also, technically, in this broad approach to error reporting, try-except blocks could have been entirely left out of the Caller code, leaving just the UI level exception handler to catch the exceptions and show a somewhat appropriate error message.

The try-except blocks in Caller code have their uses, however, in adding more detail to error messages. An attempt for a more fine-grained exception handling system was made in covariance matrix creation - see CovarianceRawDialog class and the associated Caller method create_covariance_from_raw. In this case, the short blocks in Caller code catch well defined exceptions, add a specific message to them and then raise the exceptions so that the UI code can show the actual error dialog, i.e in Caller method create_covariance_from_raw there would code like be:

```
try:
    mne.write_cov(filePathToWrite, cov)
except IOError as err:
    err.message = 'Could not write covariance file. ' + \
    'The error message was: \n\n' + err.message
    raise
```

and in CovarianceRawDialog:

```
try:
    self.caller.create_covariance_from_raw(pdict)
except IOError as e:
    self.messagebox = messageBoxes.\
        shortMessageBox(str(e.message) + str(e))
    self.messagebox.show()
    return
```

A very good question is whether or not there is any point in having an error handling system with this level of detail. If the point is to give the user the ability to tell what functionality caused the error, so that the source is known on a general level and can be e.g. added as a title to error report email, a broader system with a stack trace (and parameters used to call the method) might well be sufficient. It is questionable whether the user would needs any more detailed user-friendly messages about the source of the error; also, for the developers, the stack trace and the parameters would definitely include all the data that could be got via the messages. From the developer standpoint, adding Caller code that is not directly related to the actual logic of the task at hand also makes the method harder to read and to maintain.

In addition to local handling of exceptions, a global exception handler was implemented. Thanks to it, information for all uncaught exceptions can be shown in an error dialog, instead of just outputting the information to command line and failing silently in the UI, confusing the user.

"File reading and writing should be handled by a specific module or there should be some other way of keeping long sections of I/O code out of other program code."

This was implemented by simply expanding the existing fileManager module and adding all I/O related code to it as static methods. This made the module rather long, though, over 700 lines, as the module already contained general pickling and unpicking methods, methods related to reading and writing epochs etc. Splitting the module to a submodule for source modeling operations and a submodule for other operations might have been an option.

5.6.1 Notes on the Architecture And Testability of Meggie

The overall architecture of Meggie was not significantly altered while implementing the source analysis functionality. However, there were some architectural styles, as described by Bass et al., that were followed and constantly considered during the implementation.

The most pervading architectural style in Meggie is arguably the object oriented style and its *layered* substyle: there is separate code for the UI, user input validation, computation (the Caller class) and low level file management (the FileManager module). This layered structure can be clearly seen in the various sequence diagrams describing the implementation

of the new system features in section 5.3 - the diagrams generally have the high level UI classes or modules on the left side and the lower level ones on the right. The diagrams also show that the possible problem of the layered architectures, i.e. *layer bridging* was mostly avoided: each layer generally only calls the next layer and not the ones behind it. Bass et al. argue that this makes the modifying each layer easier - which was an explicit goal and no accident, as the previously evaluated requirements related to quality attributes are all about isolating certain types of functionality to appropriate classes or modules. Realizing these requirements should result in greater separation of concerns in the program code, hopefully yielding what Bass calls *general maintainability* - i.e. better modifiability in the face of yet unknown future modifications. (Bass, Clements, and Kazman 2003, 100-101, 259-260)

There are other styles and substyles that were considered, too. The most obvious one was the data flow style, the substyles of which are *pipe-and-filter* the *batch sequential* styles: the command line scripts the Meggie is meant to replace can be considered the processing steps of the batch sequential substyle or the filters of the pipe-and-filter substyle. When deciding the functional requirements of the source modeling implementation, an important question was when to break the simple sequential batching with interactivity - i.e. when to give the user the opportunity to examine the current state of the data in the flow (see the not-realized option the examine the reconstructed MRI image files in the source modeling preparation tab) or to branch the data flow by calling the next steps with different combinations of parameters (see the lists in forward model creation, coregistration and forward solution creation tabs). The longest streak of noninteractive batching in source modeling is the code for the forward model creation, combining three different sequential MNE methods or scripts into single functionality of Meggie. (Bass, Clements, and Kazman 2003, 96-97)

Arguably, a concern related to code and architectural quality and maintainability of the post-thesis-work Meggie is the lack of testing for the code. The external MNE-Python modules have unit tests of their own, but Meggie doesn't have any. The absence of test and testability related requirements in the requirements list explains this, but considering the focus on maintainability and the practical industry fact that testable code tends to be more maintainable, some examination is in order[3]. Again, a definition from Bass et al., backed up by Ammann

3. Interestingly, neither of the sources referenced here seems to make obvious claims about the relation of

Offutt: for the system to be testable, the inputs of the components of the system should be controllable and the corresponding outputs observable. According to Bass et al., Proper separation of concerns is also beneficial. (Bass, Clements, and Kazman 2003, 85; Ammann and Offutt 2016, 36),

The main domain logic class - and arguably the most important class to test - in Meggie is the Caller. Based on preliminary examinations, the inputs of the important Caller methods are quite controllable and and also quite separable from the user interface code, as the Caller takes only dictionaries as parameters. These parameters could well be input in tests, instead of using the input dialogs. Testing the Caller methods would still require mocking the MainWindow class, as there is a direct reference to it from the Caller and the Caller generally communicates with the MainWindow after is has successfully completed an operation and there is new information to be shown in the UI. The boundary of the Caller tests could possibly be cut there, with the possible added bonus of testing that the Caller actually communicates properly with the (mocked) MainWindow class. There is, however, also a disturbing dependency on messagebox classes for showing exception data in the UI - some solution based on e.g. Qt signals-and-slots style might be helpful with the decoupling, i.e. making sure that a Caller tests only needs check for the signal sent, not whether there actually is any reaction to it from any UI class (t. Qt Company 2016). As a bonus, this would also remove duplicated messagebox code from the Caller.

The coupling between UI and domain logic code is one of the testability concerns in Meggie - another one is the coupling between domain logic (the Caller class) and code handling the file system reads and writes (the static FileManager module). In the source modeling code of the Caller, the FileManager methods are called often, mainly to create directories to write the output datafiles to and to create the parameter files for the called MNE commands and

maintainability and testability, using these terms directly, even though Ammann Offutt do make a case about the effect of testing on the work required to change a system (see Ammann and Offutt 2016, 54-63). As for architecture and testability, though, whe could speculate that even if testable code really is generally more maintainable, it may well be because testable code simply tends to have more and better tests, not because both testability and maintainability are both caused by common architectural soundness of the system. To stay of the safe side, we could simply say that examining an architecture from the point of view testability can provide an provide an extra viewpoint that may yield useful insights to architecture in itself

methods. These methods could possibly be mocked, so that nothing is actually written to or read from the disc, and the tests could only verify that the methods are called with the correct parameters.

The testability concerns above are related to unit tests for public methods and private methods used by these public methods. A defining characteristic of unit tests is that they are limited to testing small units, in this case methods, and possibly interaction between the units and the other units directly related to them - longer chains of interaction are the domain of integration or acceptance testing. (Ammann and Offutt 2016, 23-24)

In the case of Meggie, integration testing could test that the file structures related to called methods are actually created on the disk, or that the UI reacts correctly to the signals from the Caller - the exception handling in source modeling code is designed to make sure that there is are no signals sent to the UI if the disk write fails, but there are actually no tests for this. Correspondingly, the end-to-end tests in Meggie would need to test the whole chain from the disc writes to UI reactions, and possibly the full source modeling pipeline as the user sees it, using realistic input files. Obviously, there could also be a possibility of testing the pipeline without taking the UI into account, i.e. calling the Caller methods with parameter dictionaries and checking that the correct datafiles are found in correct directories, with correct parameter files created. This level of testing requires designing a testing architecture in its own right, however, and discussing the practicalities of such architecture is definitely out of scope for the thesis work.

6 Conclusions

Meggie is a desktop application for preprocessing, visualizing and analyzing MEG data. As part of the thesis work, the feature set of the application was extended towards source analysis functionality. The work was rather exploratory and the full process chain up to source visualization was not completed. Also, no automated tests were written for the new code, nor was extensive user acceptance testing performed.

The extension work was considered a software artifact within the paradigm of design science, and was documented and evaluated as such. The main viewpoints for evaluation were code quality (especially readability and modifiability) and user interface usability. The evaluation was performed simultaneously with the documentation of the solution for each requirement defined for the software artifact.

While evaluating, the interfaces between the domain logic code in Meggie and the MNE code and scripts were found to be the most challenging of part of the system from the code quality point of view. Implementing the integration between the object-oriented Meggie code and the older MNE scripts that write directly to disk required constant search for solutions and refactoring the related code. Keeping the user interface state up-to-date with the state of the actual data was also challenging.

As for the user interface usability, the added functionality considerably increased the number of visible elements in the UI even though the basic structure remained the same. There were measures taken to keep the user interface usable nevertheless, and according to evaluation, these measures were at least partially successful.

Lastly, it should be noted that the evaluation was based on the state of the source analysis functionality as of spring 2015. The code has likely been greatly altered since then, and the thesis should be taken as an example of design science research rather than as documentation of the current state of Meggie application. Still, the thesis may be of some use for developers of MEG analysis software and scientific data analysis software in general, especially those tasked to integrate the functionality of software packages not designed to be perfectly integrated.

Bibliography

Ammann, Paul, and Jeff Offutt. 2016. *Introduction to Software Testing*. Cambridge: Cambridge University Press. ISBN: 9781107172012. https://doi.org/10.1017/9781316771273.

Bandettini, P A. 2009. "What's new in neuroimaging methods?" *Ann N Y Acad Sci* 1156 (): 260–293. https://doi.org/10.1111/j.1749-6632.2009.04420.x.

Bass, L., P. Clements, and R. Kazman. 2003. *Software Architecture in Practice*. SEI series in software engineering. Addison-Wesley. ISBN: 9780321154958.

Dale, Anders M., Bruce Fischl, and Martin I. Sereno. 1999. "Cortical Surface-Based Analysis: I. Segmentation and Surface Reconstruction". *NeuroImage* 9 (2): 179–194. ISSN: 1053-8119. https://dx.doi.org/10.1006/nimg.1998.0395.

David, Olivier, James M. Kilner, and Karl J. Friston. 2006a. "Mechanisms of evoked and induced responses in MEG/EEG". *NeuroImage* 31 (4): 1580–1591. ISSN: 1053-8119. https://dx.doi.org/10.1016/j.neuroimage.2006.02.034.

———. 2006b. "Mechanisms of evoked and induced responses in MEG/EEG". *NeuroImage* 31 (4): 1580–1591. ISSN: 1053-8119. https://dx.doi.org/10.1016/j.neuroimage.2006.02.034.

Drongelen, Wim van. 2007. *Signal processing for neuroscientists : introduction to the analysis of physiological signals*. ISBN: 0123708672.

Erich, Gamma., Richard Helm, Ralp Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. ISBN: 0-201-63361-2.

FreeSurfer Team, The. 2008. "FreeSurfer Basic Reconstruction". http://surfer.nmr.mgh.harvard.edu/fswiki/BasicReconstruction.

———. 2017. "FreeSurfer Analysis Pipeline Overview". http://surfer.nmr.mgh.harvard.edu/fswiki/FreeSurferAnalysisPipelineOverview.

Gnome Project, The. 2014. “Gnome user interface guidelines”. https://developer.gnome.org/hig/stable.

Gramfort, Alexandre. 2013. “MEG and EEG data analysis with MNE-Python”. *Front. Neurosci.* 7. ISSN: 1662-453X. https://dx.doi.org/10.3389/fnins.2013.00267.

Gramfort, Alexandre, Martin Luessi, Eric Larson, Denis A. Engemann, Daniel Strohmeier, Christian Brodbeck, Lauri Parkkonen, and Matti S. Hämäläinen. 2014. “{MNE} software for processing {MEG} and {EEG} data”. *NeuroImage* 86:446–460. ISSN: 1053-8119. https://dx.doi.org/10.1016/j.neuroimage.2013.10.027.

Gramfort, Alexandre, Martin Luessi, Eric Larson, Denis Engemann, Daniel Strohmeier, Christian Brodbeck, Roman Goj, et al. 2013. “MEG and EEG data analysis with MNE-Python”. *Frontiers in Neuroscience* 7:267. ISSN: 1662-453X. https://doi.org/10.3389/fnins.2013.00267.

Gross, J, S Baillet, G R Barnes, R N Henson, A Hillebrand, O Jensen, K Jerbi, et al. 2013. “Good practice for conducting and reporting MEG research”. *Neuroimage* 65 (): 349–363. https://doi.org/10.1016/j.neuroimage.2012.10.001.

Hansen, Peter, Morten Kringelbach, and Riitta Salmelin. 2010. “MEG: An Introduction to Methods”. https://dx.doi.org/10.1093/acprof:oso/9780195307238.001.0001.

Hari, Riitta, Lauri Parkkonen, and Cathy Nangini. 2010. “The brain in time: insights from neuromagnetic recordings”. *Annals of the New York Academy of Sciences* 1191 (1): 89–109. ISSN: 1749-6632. https://dx.doi.org/10.1111/j.1749-6632.2010.05438.x.

Hari, Riitta, and Riitta Salmelin. 2012. “Magnetoencephalography: From SQUIDs to neuroscience: Neuroimage 20th Anniversary Special Edition”. *NeuroImage* 61 (2): 386–396. ISSN: 1053-8119. https://dx.doi.org/10.1016/j.neuroimage.2011.11.074.

Hauk, Olaf, Daniel G. Wakeman, and Richard Henson. 2011. “Comparison of noise-normalized minimum norm estimates for MEG analysis using multiple resolution metrics”. *NeuroImage* 54 (3): 1966–1974. ISSN: 1053-8119. https://doi.org/10.1016/j.neuroimage.2010.09.053.

Hevner, A. R., S. T. March, J. Park, and S. Ram. 2004. “Design Science in Information Systems Research”. *MIS Quarterly* 28 (1): 75–106.

K. Aliranta, J. Pesonen. 2013. *Meggie Application Report.* Technical report 1.0.0. Accessible via the author of the thesis. Tietotekniikan laitos, Jyväskylän yliopisto.

Kruchten, P. B. 1995. “The 4+1 View Model of architecture”. *IEEE Software* 12 (6): 42–50.

Lins, Otavio G., and Terence W. Picton. 1995. “Auditory steady-state responses to multiple simultaneous stimuli”. *Electroencephalography and Clinical Neurophysiology/Evoked Potentials Section* 96 (5): 420–432. ISSN: 0168-5597. https://dx.doi.org/10.1016/0168-5597(95)00048-W.

Mitra, P P, and B Pesaran. 1999. “Analysis of dynamic brain imaging data”. *Biophys J* 76, number 2 (): 691–708. https://doi.org/10.1016/S0006-3495(99)77236-X.

MNE Developers, The. 2015. “MNE Cookbook”. http://martinos.org/mne/stable/manual/cookbook.html.

———. 2016. “MNE C Reference”. http://martinos.org/mne/stable/manual/c_reference.html.

———. 2018a. “MNE-Python source code repository”. https://github.com/mne-tools/mne-python.

———. 2018b. “The Minimum-norm Current Estimates”. https://martinos.org/mne/stable/manual/source_localization/inverse.html.

MSDN. 2010. “Windows User Experience Interaction Guidelines”. https://msdn.microsoft.com/en-us/library/windows/desktop/dn688964(v=vs.85).aspx.

Nielsen, Jacob. 1995. “10 Usability Heuristics for User Interface Design”. https://www.nngroup.com/articles/ten-usability-heuristics/.

Nielsen, Jakob. 1993. *Usability Engineering*. Cambridge, MA: Academic Press. ISBN: 0-12-518405-0.

Qt Company, The. 2016. “Qt 4.8 QMainWindow”. http://doc.qt.io/archives/qt-4.8/qmainwindow.html.

Qt Company, the. 2016. “Qt signals and slots”. http://doc.qt.io/archives/qt-4.8/signalsandslots.html.

Qt Project, the. 2018. “Qt Model View Programming”. http://doc.qt.io/archives/qt-4.8/model-view-programming.html.

Schoffelen, JanMathijs, and Joachim Gross. 2009. “Source connectivity analysis with MEG and EEG”. *Human Brain Mapping* 30. https://doi.org/10.1002/hbm.20745.

Summerfield, Mark. 2007. *Rapid GUI Programming with Python and Qt : the Definitive Guide to PyQt Programming*. ISBN: 978-0-13-235418-9.

Tallon-Baudry, C, and O Bertrand. 1999. “Oscillatory gamma activity in humans and its role in object representation”. *Trends Cogn Sci* 3, number 4 (): 151–162. http://www.ncbi.nlm.nih.gov/pubmed/10322469.

Taulu, S., M. Kajola, and J. Simola. 2003. “The Signal Space Separation method”. *(Biomed. Tech.), (To appear in Proceedings of 14th Conference of the International Society for Brain Electromagnetic Topography (ISBET)),* number 48 ().

Tiège, Xavier De, Marc Op de Beeck, Michael Funke, Benjamin Legros, Lauri Parkkonen, Serge Goldman, and Patrick Van Bogaert. 2008. “Recording epileptic activity with {MEG} in a light-weight magnetic shield”. *Epilepsy Research* 82 (2–3): 227–231. ISSN: 0920-1211. https://dx.doi.org/10.1016/j.eplepsyres.2008.08.011.

Ubuntu Community, The. 2016. “Dash As Bin Shell”. https://web.archive.org/web/20161107154000/https://wiki.ubuntu.com/DashAsBinSh.

Uusitalo, M. A., and R. J. Ilmoniemi. 1997. “Signal-space projection method for separating MEG or EEG into components”. *Medical and Biological Engineering and Computing* 35 (2): 135–140. ISSN: 0140-0118. https://dx.doi.org/10.1007/BF02534144.

Wieger, Karl. 2013. *Software Requirements 3.* Redmond: Microsoft Press, U.S. ISBN: 978-0-7356-7966-5.